國立臺北科技大學
National Taipei University of Technology

資訊工程系碩士班

碩士學位論文

Department of Computer Science &

Information Engineering

Master Thesis

低開銷斷點劃分式最早截止時間優先

Low-Overhead Breakpoint Partitioned Earliest Deadline First（LOBP-EDF）

研究生：林鈺翔

Researcher: Yu-Siang Lin

指導教授：陳香君 博士

Advisor: Shiang-Jiun Chen, Ph.D.

August 2026

國立臺北科技大學
研究所碩士學位論文口試委員會審定書

本校 資訊工程 研究所 林鈺翔 君

所提論文，經本委員會審定通過，合於碩士資格，特此證明。

學位考試委員會

委　　員：吳永庭

馬奕薇

陳香君

指導教授：陳香君

所　　長：劉建宏

中 華 民 國　一百一十五　年　7　月　28　日

摘要

關鍵詞：即時系統、劃分式最早截止時間優先、排程演算法

彈性排程是一種用於週期性即時任務的強大模型，在動態任務調整期間能有效管理系統過載，同時保證系統的可排程性。該模型最初由 Buttazzo 等人提出，允許週期性任務根據彈性參數壓縮其利用率，以因應系統壓力。然而，原始演算法具有二次時間複雜度且僅限於單處理器平台。為了適應多核心架構，Orr 等人提出了一種疊代方法將此模型擴展至多處理器排程。雖然該方法能收斂至接近精確的解，但其逐次遞增的壓縮步驟大幅增加了執行開銷。隨後，Sudvarg 等人進行了改進，提出了單處理器的準線性時間精確解，以及用於多處理器劃分式 EDF 排程的二元搜尋法。儘管如此，其單處理器演算法仍存在高昂的初始化開銷，而多處理器二元搜尋法在精準度與效率上仍有局限。 為了改善上述缺陷，本研究提出兩項貢獻。首先，針對單處理器系統的準線性時間演算法進行優化，顯著加快其執行速度，相較於基準方法實現了高達兩倍的效能提升。其次，針對多處理器環境，本研究提出「低開銷斷點劃分式最早截止時間優先框架」，其基於二元搜尋的斷點方法，並依據應用情境的需求設計出兩種模式：緊急模式採用混合斷點演算法，以最短的決策延遲完成壓縮，其執行速度較原始二元搜尋法提升可達數倍，代價為些微的過度壓縮；一般模式則採用 LB 增強型二元搜尋演算法，該演算法借用前述單處理器演算法的解析特性推導出壓縮參數的下界，藉以縮減搜尋範圍，並在維持與原始方法相當壓縮精度的前提下提供穩定的效能改善。模擬結果表明，兩種模式的可排程性判定完全一致，使系統得以依據即時性需求在決策速度與壓縮品質之間進行權衡。

TAIPEI TECH

ABSTRACT

Keywords: Real-Time Systems, Partitioned Earliest Deadline First, scheduling algorithm

Elastic scheduling is a model designed for periodic real-time tasks, highly effective in managing system overloads during dynamic task adaptation while still guaranteeing system schedulability.
The model was first proposed by Buttazzo et al. , allowing periodic tasks to compress their utilization based on an elasticity parameter in response to the pressure they are under. However, the original algorithm exhibits a quadratic-time complexity and remains restricted to uniprocessor platforms. To adapt to multi-core architectures, Orr et al. extended this model to multiprocessor scheduling by proposing an iterative algorithm. While this approach converges to a near-exact solution, its incremental compression steps significantly escalate execution overhead. Subsequently, Sudvarg et al. addressed these performance bottlenecks by introducing a quasilinear-time exact solution for uniprocessors and a binary search approach for multiprocessor partitioned earliest deadline first scheduling. Despite these advancements, the uniprocessor algorithm incurs a substantial initialization overhead, while the multiprocessor binary search approach suffers from limitations in efficiency.

To address these shortcomings, this study presents two primary contributions. First, we optimize the quasilinear-time algorithm for uniprocessor systems, drastically accelerating its execution speed to achieve up to a two-fold performance improvement over the baseline approach. Second, for multiprocessor environments, we propose the Low-Overhead Breakpoint Partitioned Earliest Deadline First (LOBP-EDF) framework. This framework is built upon a binary-search-based breakpoint method and offers two operating modes tailored to varying application scenario requirements. Specifically, Emergency Mode deploys the Hybrid Breakpoint Search algorithm to complete utilization compression with minimal decision latency, achieving a speedup of up to several fold compared to the original binary search at the cost of slight over-compression. Conversely, Normal Mode employs the LB-Enhanced Binary Search algorithm, which leverages the analytical properties of the aforementioned uniprocessor algorithm to derive a lower bound for the compression parameter. This lower bound successfully narrows the search range, providing stable performance improvements while maintaining compression accuracy comparable to the original method. Simulation results demonstrate that the schedulability decisions of

TAIPEI TECH

both modes are completely identical, thereby enabling the system to flexibly trade off decision speed against compression quality based on dynamic real-time requirements.



Acknowledgements

在碩士論文完成之際，首先我要感謝我的指導教授陳香君博士。

從我確定研究主題之初，便給予我滿滿的肯定與鼓勵；在研究期間提供了非常多的指導與建議；到最後的論文撰寫與定稿，也耐心地為我審閱與修正，讓我得以順利完成這篇論文並畢業。

再來，感謝擔任口試委員的吳和庭博士與馬奕葳博士。感謝兩位老師在繁忙中抽空審閱本篇論文，並在口試時給予建言與指正，讓我得以發現並修正論文的不足之處，使這篇論文更加完善。

接著，感謝國立臺北科技大學資訊工程系軟體安全實驗室的大家。不論是碩班繁重的課業、研究上的難題，還是最後的畢業流程，你們都給了我非常多的幫助。另外要感謝我的朋友們，栗言、承翰、晨閔、益德、榕晟、辰允，謝謝你們從高中開始一路的陪伴，不論是在平日課後陪我玩遊戲、排解寂寞，還是在我難過時幫我散心。

最後，我要感謝我的爸爸、媽媽和哥哥。感謝你們從小到大無微不至的照顧與包容；有了你們全方位的支持，才讓我能夠沒有後顧之憂地全力攻讀碩士。

林鈺翔 謹誌於

台北科技大學資訊工程系碩士班

中華民國 115 年 7 月 28 日

TAIPEI TECH

Table of Contents

摘要 i

ABSTRACT ii

Acknowledgements iv

Table of Contents v

List of Tables vii

List of Figures viii

Chapter 1 Introduction 1

1.1 Motivation and Background 1

1.2 Objective and Contribution 2

1.3 Organization 3

Chapter 2 Related Work 4

2.1 Real-Time Scheduling Paradigms 4

2.1.1 Uniprocessor Elastic Scheduling 4

2.1.2 Multiprocessor Elastic Scheduling 10

2.2 Allocation Heuristics and Evaluation Metrics 13

2.2.1 Allocation Heuristics 13

2.2.2 Evaluation Metrics 14

2.3 Generation Methods for Real-Time Task Sets 15

2.3.1 UUnifast 17

2.3.2 UUnifast-Discard 17

2.3.3 RandFixedSum 17

2.3.4 Dirichlet-Rescale (DRS) 18

Chapter 3 Methodology 19

3.1 Main Method 19

3.1.1 Improved Elastic Scheduling Algorithm 21

TAIPEI TECH

3.1.2 Hybrid Breakpoint Search Algorithm 22
3.1.3 LB-Enhanced Binary Search Algorithm 25
3.2 Pseudocode and Detailed Algorithm Description 26
Chapter 4 Experiment and Results 32
4.1 Requirements 32
4.2 Environment Setup 33
4.3 Experiment 36
4.3.1 Task Sets Generation 36
4.3.2 Experimental Evaluation 37
Chapter 5 Conclusion and Future Work 50
5.1 Conclusion 50
5.2 Future Work 51
References 53

TAIPEI TECH

List of Tables

Table 4.1 Software Requirements 32

Table 4.2 Third-Party Software Requirements 33

Table 4.3 Hardware Requirements 33

Table 4.4 Mean, median, and maximum execution cycles of uniprocessor algorithms during the compression phase (N=50) 40

TAIPEI TECH

List of Figures

Figure 3.1 Flowchart of the proposed LOBP-EDF framework 20

Figure 3.2 Flow Chart of Hybrid Breakpoint Search Algorithm 24

Figure 4.1 Raspberry Pi Imager Interface 33

Figure 4.2 Uniprocessor Algorithms Result Comparison 38

Figure 4.3 Uniprocessor Algorithms Compression Time Comparison 40

Figure 4.4 Multiprocessor Algorithms Comparison M=4 43

Figure 4.5 Multiprocessor Algorithms Comparison M=8 44

Figure 4.6 Multiprocessor Algorithms Comparison M=16 45

Figure 4.7 Lambda difference between LB-Enhanced and Original Binary 48

Figure 4.8 Lambda difference between BP_Hybrid and Original Binary 49

TAIPEI TECH

Chapter 1 Introduction

1.1 Motivation and Background

Elastic scheduling is an elegant model designed for periodic real-time tasks. It is particularly effective in managing system overloads during dynamic task adaptation while still guaranteeing system schedulability. First introduced by Buttazzo et al. [1], this model allows periodic tasks to compress their utilization based on an elasticity parameter in response to system pressure. The original elastic task model was primarily tailored for preemptive uniprocessors. Under this model, each task is characterized by a worst-case execution time (WCET) and an acceptable period range, which are governed by three primary parameters: a maximum utilization parameter, an elasticity coefficient, and a minimum utilization parameter. To maintain the total utilization of the task set below the desired threshold, tasks must be assigned periods within their acceptable ranges. The original algorithm, however, suffers from a quadratic time complexity of $O(n^2)$ when performing admission control or adjusting to changes in the utilization bound. This performance bottleneck not only hampers its scalability but also confines the algorithm to single-processor platforms. As real-time systems transitioned to multi-core platforms, Orr et al. [2] extended the original quadratic-time model to multiprocessor scheduling, proposing an iterative algorithm for sequential tasks on homogeneous multi-core systems. This algorithm incrementally increases the utilization compression applied to the tasks until the system deems the task set schedulable. Although this iterative approach converges to a near-exact solution, the incremental steps significantly escalate the overall execution time. The authors subsequently consolidated and extended this line of work in a journal version [3], which provides a comparative evaluation of several algorithms for implementing elastic tasks on multiprocessor platforms under both global and partitioned paradigms.

To address these performance bottlenecks, Sudvarg et al. [4], [5] made substantial improvements. For uniprocessor systems with static utilization bounds, the authors proposed a near-

TAIPEI TECH

exact solution algorithm operating in quasilinear time. Furthermore, they introduced two distinct approaches for multiprocessor partitioned earliest deadline first scheduling. Altogether, their work introduces three core algorithms, each possessing unique characteristics and optimization opportunities that directly motivate this study.

- The Uniprocessor Quasilinear Algorithm: Despite its theoretical quasilinear complexity, this algorithm suffers from a high initialization overhead. This substantial constant factor prevents it from yielding significant practical advantages over the original $O(n^2)$ approach. Therefore, further reducing its execution overhead is imperative to unlock its full performance potential.
- The Heuristic-Based Multiprocessor Algorithm: As an extension of the uniprocessor quasi-linear algorithm to multiprocessor environments, this approach operates under dynamic utilization bounds—a scenario where Buttazzo's original algorithm likewise cannot be directly applied. It leverages specific allocation heuristics to establish schedulability bounds. Although the resulting analysis remains somewhat pessimistic, this heuristic approach yields an exceptionally high speedup factor, enabling the rapid identification of approximate solution. This makes it particularly well-suited for scenarios demanding fast mode switches, such as mixed-criticality systems. Given its high execution efficiency, this study does not directly modify this algorithm; however, the core concept of utilizing allocation heuristics to establish bounds will be integrated into our subsequent methodology.
- The Binary-Search Based Multiprocessor Algorithm: This approach incorporates binary search into partitioned EDF scheduling as an improvement over Orr's iterative algorithm. However, significant room remains to develop a more advanced approach that locates a near-exact solution more rapidly, thereby achieving greater performance gains and preventing resource waste.

1.2 Objective and Contribution

Building upon the strengths of existing literature while addressing their limitations, this study delivers two primary contributions:

- Uniprocessor Performance Doubling: We optimize the quasilinear-time algorithm for uniprocessor systems, significantly accelerating its execution speed to achieve up to a two-fold performance gain over the baseline approach.
- Context-Aware Multiprocessor Acceleration Framework: For multiprocessor environments, we propose the Low-Overhead Breakpoint Partitioned Earliest Deadline First (LOBP-EDF) framework. Built upon a binary-search-based breakpoint method, this framework is dynamically tailored to different operational scenarios by offering two distinct modes. Emergency Mode: Employs the Hybrid Breakpoint Search algorithm to drastically reduce decision latency through discrete search; Normal Mode: Utilizes the LB-Enhanced Binary Search algorithm, which leverages the analytical properties of the aforementioned uniprocessor algorithm to derive a lower bound for the compression parameter, thereby narrowing the search range while maintaining compression accuracy comparable to the original approach.

1.3 Organization

The remainder of this thesis is organized as follows. Chapter 2 reviews the foundational concepts of real-time scheduling, tracing the evolution of the elastic task model from its traditional roots to subsequent refinements and multi-core extensions. Additionally, it outlines the methods for generating synthetic datasets, task allocation heuristics, and evaluation metrics. Chapter 3 details the design and implementation of the proposed Low-Overhead Breakpoint Partitioned Earliest Deadline First (LOBP-EDF) methodology, including the quasilinear algorithm for uniprocessors and the breakpoint algorithm for multiprocessors. Chapter 4 presents the experimental setup, followed by a comprehensive performance analysis and comparison. Finally, Chapter 5 summarizes the main findings of this study and outlines potential directions for future research.

TAIPEI TECH

Chapter 2 Related Work

This chapter presents the foundational concepts and related work. We begin by examining single-processor and multiprocessor scheduling paradigms. Next, we review the allocation heuristics and evaluation metrics employed across these platforms. Finally, we introduce the methodologies for generating synthetic task sets used to evaluate real-time schedulability tests.

2.1 Real-Time Scheduling Paradigms

This section surveys relevant scheduling methodologies. We first examine the foundational elastic model for uniprocessor systems and its evolutionary improvements in Section 2.1.1. Section 2.1.2 extends this review to the multiprocessor domain, focusing specifically on partitioned EDF scheduling, the final application scenario for the approach proposed in this study.

2.1.1 Uniprocessor Elastic Scheduling

When applying the elastic model to a real-time control system, there is no guarantee that a new task will be accepted if the system is overloaded. In such cases, the system can leverage this model [1] to reduce the utilization of existing tasks—effectively increasing their periods—to lower the total load and make room for the new task, rather than rejecting it outright; such periodic tasks are designed as cyclical processor activities; therefore, each periodic task instance, τ_i, has two constant parameters: computation time C_i and period T_i. The concept of the elastic model is analogous to treating a task as a spring with a stiffness coefficient and length constraints. Each task is characterized by five parameters as an elastic task: $\tau_i(C_i, T_{i_0}, T_{i_{\min}}, T_{i_{\max}}, e_i)$, where C_i is the computation time; T_{i_0} is the nominal period, representing the execution interval under ideal conditions; the system operates based on this value during normal operation; $T_{i_{\min}}$ is the minimum period, representing the lower limit for maintaining task execution; $T_{i_{\max}}$ is the maximum period, representing the upper limit to which the task's period can be extended during overload; e_i is the elastic coefficient, indicating the flexibility of a task to adjust its own utilization to adapt to a new feasible rate configuration; a larger value

indicates greater elasticity and the ability to compress the period further. Any change to a task must comply with the elastic guarantee and is only permitted if there exists a feasible schedule in which all other periods fall within the task's range. Next, we explain how the elastic model achieves the elastic guarantee. At any given time, the set of periodic tasks Γ can be divided into two subsets: Γ_f, consisting of fixed tasks with minimum utilization; and Γ_v, consisting of tasks that can be further compressed. Let $U_{i_0} = \frac{C_i}{T_{i_0}}$ be the nominal utilization of τ_i; U_0 is the sum of all nominal utilizations; U_f is the total utilization factor of the tasks in Γ_f; U_d is the desired utilization, representing the target total utilization the system aims to achieve; and to achieve the required utilization $U_d < U_0$, each task must be compressed to a utilization below:

$$\forall \tau_i \in \Gamma_v \quad U_i = U_{i_0} - \left(U_0 - U_d + U_f\right)\frac{e_i}{E_v} \tag{2.1}$$

where:

$$U_f = \sum_{\tau_i \in \Gamma_f} U_{i_{\min}} \tag{2.2}$$

$$E_v = \sum_{\tau_i \in \Gamma_v} e_i \tag{2.3}$$

when a task yields $U_i < U_{i_{\min}}$, this means that the period of these tasks must be fixed at their maximum value $T_{i_{\max}}$ that is, $U_i = U_{i_{\min}}$, then update U_f and E_v, as well as Γ_f and Γ_v in the subsequent process. Afterward, it is only necessary to apply Equation 2.1 iteratively to the tasks in Γ_v until a feasible solution is found that is, until the required utilization U_d is greater than or equal to the minimum possible utilization $U_{\min} = \sum_{i=1}^{n} \frac{C_i}{T_{i_{\max}}}$.

Algorithm 1: Task compress (Adapted from [1, Fig. 3].)

```
1: function TASK COMPRESS(Γ, U_d)
2:     U_0 ← Σ_{i=1}^{n} C_i/T_{i_0}
3:     U_min ← Σ_{i=1}^{n} C_i/T_{i_max}
4:     if U_d < U_min then
5:         return infeasible
6:     end
7:
```

```
8:    while true do
9:        U_f ← 0
10:       E_v ← 0
11:       for each τ_i do
12:           if e_i == 0 or T_i == T_{i_max} then
13:               U_f = U_f + U_i
14:           else
15:               E_v = E_v + e_i
16:           end
17:       end
18:   end
19:
20:   ok = 1
21:   for each τ_i ∈ Γ_v do
22:       if e_i > 0 and T_i < T_{i_max} then
23:           U_i = U_{i_0} − (U_0 − U_d + U_f) e_i / E_v
24:           T_i = C_i / U_i
25:           if T_i > T_{i_max} then
26:               T_i = T_{i_max}
27:               ok = 0
28:           end
29:       end
30:   end
31:   if ok ≠ 0 then
32:       break
33:   end
34:   end
35:   return feasible
36: end
```

Algorithm 1 determines the schedulability of a set of overloaded flexible tasks Γ after compression. Initially, Lines 4–6 check whether the target utilization $U_d < U_{\min}$ is satisfied; if so, the algorithm returns unschedulable because scheduling is not possible without exceeding the system's lower limit. Lines 9–10 initialize the algorithm. Lines 11–18 check the conditions for and compute τ_i. Lines 12–16 accumulate the U_i values of tasks in Γ_f which are inelastic variables or have reached their own upper limits into U_f if they satisfy $e_i = 0$ or $T_i = T_i^{\max}$;

TAIPEI TECH

For those that do not satisfy the conditions and belong to Γ_v, the algorithm accumulates their e_i into E_v; the flag in line 20 is used to check and control the iteration; Lines 21–22 run Line 23 (Equation 2.1) on the re-compressible subset of Γ_v; Line 24 calculates their T_i; and Lines 25–27 check the flag. If it is not set, the next iteration continues. When the iteration ends, "feasible" is returned to the main program.

Sudvarg et al. [4], [5] pointed out that the original elastic task model is designed to perform online adaptation to accommodate dynamic and open systems, where handling task admission requires high computational efficiency to maintain the quality of service. To address scheduling scenarios governed by a utilization bound—where the total utilization is evaluated against a fixed threshold (e.g., a bound of 1 ensuring 100% safety on a uniprocessor system)—the authors proposed a significant improvement over the original model. Their approach can find a near-exact solution in quasilinear time under a fixed utilization bound, and can subsequently handle admission control or dynamic adjustments to the utilization bound in linear time.

To facilitate extension to a multiprocessor system, the authors equivalently transform the five parameters of the original model—specifically, the period parameters T_i, $T_{i_{\min}}$, and $T_{i_{\max}}$ into utilization parameters U_i; $U^i_{\max}$; and $U^i_{\min}$. Since $T_i = \frac{C_i}{U_i}$ and C_i is a fixed constant, knowing U_i is sufficient to determine T_i. Note that $T_{i_{\max}}$ is transformed into $U^i_{\min}$ (because, in a physical sense, the least CPU power is consumed when the period is longest), while $T_{i_{\min}}$ is transformed into $U^i_{\max}$.

After this transformation, the fundamental concept becomes: the uncompressed utilization $U^{\max}_{\mathrm{SUM}}$ of a task system $\Gamma = \{\tau_1, ..., \tau_n\}$ can be expressed as:

$$U^{\max}_{\mathrm{SUM}} = \sum_{i=1}^{n} U^{\max}_i \tag{2.4}$$

Let the desired utilization U_d be defined as the utilization bound allowed by the current algorithm; after substituting the variables, Equation 2.1 becomes:

$$U_i = U^{\max}_i - \left(\frac{U_{\mathrm{SUM}} - (U_d - \Delta)}{E_{\mathrm{SUM}}} \right) \times E_i \tag{2.5}$$

When an overload occurs, each task τ_i is reassigned a utilization U_i governed by three conditions, which are: 1. $\sum_i U_i = U_d$; 2. For any task with $E_i = 0$ is considered inelastic, meaning $U_i^{\min} = U_i^{\max}$; 3. For other tasks τ_i and τ_j, if $U_i > U_i^{\min}$ and $U_j > U_j^{\min}$, then U_i and U_j must satisfy the following relationship:

$$\left(\frac{U_i^{\max} - U_i}{E_i}\right) = \left(\frac{U_j^{\max} - U_j}{E_j}\right) \tag{2.6}$$

A Γ that satisfies all of the above conditions indicates that the system is feasible.

The improved algorithm proposed by the authors define a per-task attribute Φ_i for every elastic task τ_i:

$$\Phi_i \stackrel{\text{def}}{=} \left(\frac{U_i^{\max} - U_i^{\min}}{E_i}\right) \tag{2.7}$$

In the proof in [6], it is shown that, in the original model, the transfer of tasks from Γ_v to Γ_f can proceed in non-increasing Φ_i order at a cost of $O(n \log n)$; after the tasks are sorted so that $\Phi_i \leq \Phi_i + 1$ for every i $(1 \leq i < n)$, one linear pass over $\tau_1 \ldots \tau_n$ suffices, thereby completing all accumulation steps U_i belonging to Γ_f before computing Γ_v. The time complexity of these steps is $O(n)$, so the algorithm's time complexity is $O(n \log n)$.

Algorithm 2: Elastic_Implicit_Uniprocessor (Adapted from [5, Algo. 2].)

function ELASTIC_IMPLICIT_UNIPROCESSOR(Γ, U_d)
  Input:A list Γ of elastic tasks sorted in non-decreasing order of their Φ_i parameters
  and a desired utilization U_d
  Output: Feasibility and the list Γ with computed U_i values
  $U_{\text{SUM}} \leftarrow 0$
  $E_{\text{SUM}} \leftarrow 0$
  $\Delta \leftarrow 0$
  for $\tau_i \in \Gamma$ **do**
    $U_{\text{SUM}} = U_{\text{SUM}} + U_i^{\max}$
    $E_{\text{SUM}} = E_{\text{SUM}} + E_i$
  end
  for $\tau_i \in \Gamma$ **do**
    if $U_i^{\max} - \frac{U_{\text{SUM}} - (U_d - \Delta)}{E_{\text{SUM}}} \times E_i \leq U_i^{\min}$ **then**

14: ▷ Task τ_i is no longer compressible - it's in Γ_{FIXED}
15: $U_i \leftarrow U_i^{\min}$ ▷ Since $\tau_i \in \Gamma_{\text{FIXED}}$
16: $\Delta \leftarrow \Delta + U_i^{\min}$ ▷ This additional amount of utilization is allocated to tasks in Γ_{FIXED}
17: **if** $\Delta > U_d$ **then** ▷ Cannot accommodate the minimum requirements
18: **return** Infeasible
19: **end**
20: $U_{\text{SUM}} \leftarrow U_{\text{SUM}} - U_i^{\max}$ ▷ Since τ_i is removed from Γ_{VARIABLE}
21: $E_{\text{SUM}} \leftarrow E_{\text{SUM}} - E_i$ ▷ As above - since τ_i is removed from Γ_{VARIABLE}
22: **else**
23: ▷ Remaining tasks are all compressible (i.e. , in Γ_{VARIABLE})
24: $U_i \leftarrow U_i^{\max} - \frac{U_{\text{SUM}} - (U_d - \Delta)}{E_{\text{SUM}}} \times E_i$
25: **end**
26: **end**
27: **return** feasible
28: **end**

Algorithm 2 is a set of overloaded flexible tasks Γ that includes non-increasingly sorted Φ_i values and desired utilization U_d, it determines schedulability after compression and calculates the new compressed U_i values. Initially, Lines 5–11 constitute the initialization phase of the algorithm, calculating U_{SUM} and E_{SUM} for subsequent processing. The Compression phase is covered in lines 12–26, Line 13, Equation 2.5 is used to check whether the tasks τ_i belonging to Γ_{FIXED} can no longer be compressed; Line 15, the U_i value for τ_i is set to $U_i^{\min}$; Line 16, the U_i values from above are accumulated into Δ; Lines 17–19 check whether Δ exceeds U_d; if so, the elastic task Γ is reported as infeasible. Lines 20–21 subtract the U_i^{MAX} and E_i of τ_i belonging to Γ_{FIXED} from U_{SUM} and E_{SUM}, to avoid affecting subsequent calculations. Lines 22–25, the algorithm uses Equation 2.5 to calculate the U_i values for tasks belonging to Γ_{VARIABLE}, until the loop ends and "feasible" is returned.

The elastic scheduling work discussed above, including the model of Buttazzo et al. [1] and the improvements of Sudvarg et al. [4], [5], is restricted to implicit-deadline tasks, where each task's relative deadline equals its period. Extending elastic scheduling beyond this assumption has recently attracted attention: Baruah [7] addressed systems of sporadic constrained-deadline

elastic tasks and proposed improved uniprocessor scheduling for them, and Liu et al. [8] subsequently developed schedulability analysis and performance optimization techniques for the same task class. As this thesis builds directly upon the implicit-deadline algorithms of [4], [5], the constrained-deadline setting is left outside our scope and is revisited as a direction for future work in Chapter 5.

2.1.2 Multiprocessor Elastic Scheduling

As real-time systems have shifted from single-processor to multiprocessor architectures, flexible scheduling models have been expanded accordingly. Orr and Baruah [2], [3] point out that on a multiprocessor system, it is difficult to directly calculate the exact compression ratio using a closed-form expression, as can be done for a uniprocessor system. Partitioned scheduling, in this context, is equivalent to the bin-packing problem, for which there are several polynomial time heuristic algorithms to find approximate solutions. Most of these algorithms share the following common structure: First, the ordering of tasks is determined. Then, as each task is considered in turn, the order in which they will be attempted on each processor is determined based on the selected sequence. If a task's requirements fit into the remaining space on that processor, the task is deemed successfully assigned to that processor. The same applies in the context of partitioned-EDF scheduling: a task is placed on a core only when its utilization fits the remaining capacity — the core's capacity less the total utilization already assigned there. The procedure succeeds only if every task is placed; should any task fail to fit, it returns failure.

Drawing on Lopez et al. [9], who found that any partitioning heuristic can be adapted to period-elastic tasks, the authors propose a λ based linear search algorithm. They consider a set of period-elastic tasks, where $\Gamma = \{\tau_i = (U_i^{\max}, U_i^{\min}, E_i)\}_{i=1}^{n}$ to be scheduled on m identical unit-speed processors. Sweeping λ from 0 to Φ at resolution ε locates the smallest λ under which the chosen heuristic can partition the set onto m unit-speed processors, where each task's utilization at a given λ is assigned by:

$$U_i \leftarrow \max(U_i^{\max} - \lambda E_i, U_i^{\min}) \tag{2.8}$$

The time complexity of this algorithm for sorting and partitioning tasks is $\theta(n \times \log(n) + n \times m)$, therefore, the overall time complexity for iteratively determining the λ value in partitioned scheduling is $\theta\big(\frac{\Phi}{\varepsilon} \times (n \times \log(n) + n \times m)\big)$.

Subsequently, Sudvarg et al. [4], [5] observed an improvement to the original algorithm: rather than scanning every λ in $[0, \lambda^{\max}]$ step-by-step at granularity $\varepsilon = \frac{\lambda^{\max}}{1000}$, they replace the linear sweep with a binary search running in $\log\big(\frac{\lambda^{\max}}{\varepsilon}\big)$ time, further reducing the overall time complexity to $\theta\big((n \times \log(n) + n \times m) \times \log\big(\frac{\lambda^{\max}}{\varepsilon}\big)\big)$. The improved algorithm is described below:

Algorithm 3: Elastic_partitioned_EDF (Adapted from [5, Algo. 3].)

```
1:  function ELASTIC_PARTITIONED_EDF(Γ, m)
2:      Input: A list Γ of elastic tasks to schedule on m processor cores
3:      Output: The value λ to obtain feasiblity
4:      λ^max ← 0
5:      for τ_i ∈ Γ do
6:          λ_i^max ← (U_i^max − U_i^min) / E_i
7:          λ^max ← max(λ^max, λ_i^max)
8:      end
9:      if Γ(0) is schedulable on m cores then return 0
10:     if Γ(λ^max) is not schedulable on m cores then return INFEASIBLE
11:     λ_HI ← λ^max
12:     λ_LO ← 0
13:     while true do
14:         λ ← (λ_HI + λ_LO)/2
15:         if Γ(λ) is schedulable on m cores then λ_HI ← λ
16:         else λ_LO ← λ
17:         if not (λ_HI − λ_LO > ε) then
18:             break
19:         end
20:     end
21:     return λ_HI
22: end
```

Algorithm 3 computes, for a task set Γ, the smallest λ that makes it feasible across m cores. Initially, Line 4, $\lambda^{\max}$ is initialized, Lines 5–8 use Equations 2.7 to compute the maximum $\lambda^{\max}$ value for all τ_i in Γ; Line 9 checks the $\Gamma(0)$ indicates whether the uncompressed task set can be

TAIPEI TECH

scheduled on m cores; if the condition is satisfied, $\lambda = 0$ is returned; Line 10 checks whether the maximally compressed task set $\Gamma(\lambda^{\max})$ can be scheduled; if not, it indicates that even the largest lambda value in the entire Γ set cannot be scheduled, and "infeasible" is returned; Lines 11–12 initialize the variables used for the binary search; Lines 13–20 perform a binary search on the interval $[0, \lambda^{\max}]$: λ_{HI} is the smallest value examined so far for which $\Gamma(\lambda)$ is schedulable; Meanwhile, λ_{LO} starts at 0 and tracks the largest value of lambda that satisfies $\Gamma(\lambda)$ but is not schedulable; schedulability is checked in each iteration of the loop; If the result is feasible, the algorithm updates λ_{HI} to that value; otherwise, it updates λ_{LO} to that value. When the difference between the two variables does not exceed ε, the algorithm terminates and returns λ_{HI}.

The authors also point out that, as noted in [10] it can be seen that, under the first-fit and best-fit heuristics, a set of tasks Γ can be scheduled across m processor cores if its total utilization $\sum_i U_i = m$ does not exceed $\frac{m+1}{2}$ and the utilization of any single task does not exceed 1; this means that algorithm 2 can be applied by compressing algorithm's U_d to the safe threshold of $\frac{m+1}{2}$, thereby achieving fast task compression in $O(n \log n)$ time; however, since $\frac{m+1}{2}$ is the conservative sufficient utilization bound for which the first-fit and best-fit heuristics do not guarantee schedulability, applying this algorithm alone may result in unnecessarily aggressive compression, leading to over-compression. In contrast, although the original iterative algorithm and the subsequently improved binary search algorithm may execute more slowly, they can reduce the pessimism introduced by the sufficient utilization bound in both avoiding overly aggressive compression and successfully scheduling a larger number of task sets. We will incorporate this concept into the experiments for the algorithm described in this paper.

Beyond the sequential, implicit-deadline setting considered in this thesis, the elastic task model has been extended along several complementary dimensions. Sudvarg et al. [11] applied workload adaptation to fork-join tasks, and later refined the granularity of adaptation to the subtask level [12]. Orr et al. [13] replaced the continuous utilization range with a finite candidate set, showing that the resulting problem is NP-hard; although our discretization is applied over the compression parameter λ rather than over per-task utilizations, it shares the same underlying

idea and motivates the breakpoint method presented in Chapter 3. The model has also been applied to mixed-criticality systems, where Su and Zhu [14] introduced an elastic mixed-criticality model and Sun et al. [15] used elastic compression to achieve graceful degradation across a mode switch. These extensions are complementary to the present work, which instead focuses on reducing the execution overhead of the compression procedure itself.

To explain the sorting strategy adopted by the authors in their algorithms, as well as to supplement the classification criteria for multiprocessor task-scheduling algorithms and the theoretical metrics for evaluating their performance, two relevant references are introduced in Section 2.2.

2.2 Allocation Heuristics and Evaluation Metrics

In multiprocessor scheduling algorithms, the process of assigning tasks to individual cores is equivalent to the bin-packing problem, which is a strongly NP-hard problem. To address this complex problem, polynomial-time heuristic algorithms are commonly used to find approximate solutions. This section will explore in detail the classifications of task-assignment algorithms proposed by López et al. [9] and Baruah [10] , as well as the two core theoretical metrics used to evaluate their performance: the utilization bound and the speedup factor.

2.2.1 Allocation Heuristics

López et al. [9] define this type of task-scheduling heuristic as Reasonable Allocation (RA). A characteristic of an RA algorithm is that it declares scheduling failure only when no processor in the system can accommodate the new task. Common RA strategies include: First-Fit (FF), which assigns a task to the first processor capable of accommodating it; Best-Fit (BF), which assigns a task to the processor with the least remaining space that can meet the task's requirements; and Worst-Fit (WF), which assigns a task to the processor with the most remaining space.

Building on RA, the authors further defined the Reasonable Allocation Decreasing (RAD) class of algorithms. Before performing allocation, RAD algorithms first sort all tasks in non-increasing order based on their utilization, and then allocate them sequentially. Algorithms

belonging to this class include First-Fit Decreasing (FFD), Best-Fit Decreasing (BFD), and Worst-Fit Decreasing (WFD).

2.2.2 Evaluation Metrics

To quantify and compare the performance of these allocation algorithms, López et al. [9] conducted an in-depth study of the utilization bounds for EDF scheduling on multiprocessor systems.

- The utilization bound metric indicates that if the total utilization of a task set is below this value, the algorithm guarantees that it can successfully allocate and schedule the tasks. Assuming a platform with m processors, let α denote an upper bound on the utilization of any individual task, such that $0 < U_i \leq \alpha \leq 1$. The parameter β is then defined as $\beta = \lfloor \frac{1}{\alpha} \rfloor$.
- Mathematical proofs show that WF and Worst-Fit Increasing (WFI) attain the minimum worst-case utilization bound among the considered reasonable allocation algorithms: $m - (m - 1)\alpha$. In contrast, FF, BF, and all RAD-class algorithms achieve the theoretical upper bound of $\frac{\beta m+1}{\beta+1}$.
- If no restrictions are placed on the single utilization rate of the task—that is, if no utilization cap tighter than $U_i \leq 1$ is imposed, then α=1 and β=1. The utilization rate bounds for FF, BF, and RAD algorithms are all $\frac{m+1}{2}$. This theoretical result explains why, in Section 2.1.2, the total utilization of $\frac{m+1}{2}$ was considered a safe bound guaranteeing schedulability for heuristic methods such as First-Fit and Best-Fit; however, using this as the sole criterion would lead to unnecessary over-compression.

Although FF, BF, and RAD-type algorithms perform consistently at the utilization threshold, Baruah et al. [10] pointed out that the utilization bound remains an inadequate evaluation metric because the utilization bound cannot adequately distinguish the quality of different partitioning heuristics. Once the total utilization exceeds (m+1)/2, some task sets remain feasibly partitionable, whereas others are infeasible even under an optimal partitioning algorithm. This prevents the utilization bound from accurately distinguishing whether a scheduling failure is due to the

TAIPEI TECH

poor performance of the heuristic algorithm itself or the inherent limitations of the partition-based scheduling mechanism. To address this, the authors introduce the Speedup Factor as an additional evaluation metric.

- The speedup factor f of an algorithm is the smallest value $f \geq 1$ such that every task set feasible on m unit-speed processors is guaranteed to be successfully partitioned by the algorithm on m processors of speed f.
- Mathematical proofs conclude that the speedup factor for all RAD algorithms (such as FFD, BFD, and WFD) does not exceed $\frac{4}{3} - \frac{1}{3m}$. In contrast, the lower bound for the speedup factor of FF and BF class algorithms is $2 - \frac{2}{m+1}$; while the lower bound for the speedup factor of WF class algorithms is $2 - \frac{2}{m}$. This metric clearly shows that as the number of processors m increases, the lower bounds for the speedup factors of the unsorted FF, BF, and WF heuristics approach 2. In contrast, every RAD algorithm has a speedup factor of at most $\frac{4}{3} - \frac{1}{3m}$, which approaches 4/3.

Based on the foundational analysis above, RAD algorithms attain the maximum worst-case utilization bound and provide stronger speedup-factor guarantees than the corresponding unsorted heuristics. Consequently, these robust heuristics are selected as the core allocation strategies for the subsequent evaluations in this paper.

2.3 Generation Methods for Real-Time Task Sets

As highlighted by Griffin et al. [16], when conducting experimental evaluations of schedulability tests for real-time systems, researchers typically focus on two critical aspects: effectiveness and tractability. Generally, a tradeoff exists between these two goals.

A relatively small number of exact schedulability tests may become computationally intractable, whereas for large task sets generated through quick and simplified methods, many task sets that are actually schedulable may be falsely classified as unschedulable.

The underlying logic for generating a synthetic task set revolves around the utilization U_i of each task τ_i, which represents the fraction of processor capacity required by the task. In the

traditional sporadic task model, a task's utilization is determined by its worst-case execution time C_i and its period or minimum inter-arrival time T_i via $U_i = \frac{C_i}{T_i}$. Consequently, the total utilization U of a task set containing n tasks is calculated as:

$$U = \sum_{i=1}^{n} U_i \tag{2.9}$$

To ensure baseline schedulability, the total utilization cannot exceed the processing bandwidth provided by the system. Assuming that intra-task parallelism is not permitted, $U \leq m$ and ($U_i \leq 1$) are necessary, but not sufficient, conditions for schedulability on m identical unit-speed processors.

As further emphasized by Griffin et al. [16], task set generation must be executed carefully to avoid confounding variables. A commonly used but problematic approach is to repeatedly add tasks until the target utilization is reached. This procedure introduces a strong correlation between task-set cardinality (n) and total utilization (U). Such entanglement makes it impossible to discern whether the performance of a schedulability test is influenced by the number of tasks or by the utilization level itself. To address this issue, an ideal task utilization generation algorithm must satisfy three key requirements:

- Efficiency: The algorithm must be computationally fast enough to generate millions of task sets for extensive simulations.
- Uniformity: The generated utilization vectors must be unbiased and uniformly distributed across the valid search space.
- Flexibility: The algorithm must accommodate specific constraints on the individual utilization values of the n tasks.

In the following sections, we review four prominent methods for generating task utilization vectors. We first examine three traditional unbiased approaches widely adopted in the literature —UUnifast, UUnifast-Discard, and RandFixedSum—and analyze how they align with the above evaluation criteria. Subsequently, we introduce the Dirichlet-Rescale (DRS) algorithm, a general-purpose method designed to overcome the limitations of these traditional approaches.

TAIPEI TECH

2.3.1 UUnifast

The UUnifast algorithm, proposed by Bini and Buttazzo [17], is highly efficient but fails to meet the flexibility requirement. Its constraint vectors are strictly fixed at $u^{\max} = (U, U, ..., U)$ and $u^{\min} = u^0 = (0, 0, ..., 0)$. Because it was originally designed for single-processor system research, UUniFast can generate vectors with $U > 1$, but it does not enforce the per-task constraint $U_i \leq 1$. Consequently, it cannot be applied to multiprocessor system experiments where the total utilization exceeds one ($U > 1$).

2.3.2 UUnifast-Discard

To extend the UUnifast model to multiprocessor systems, Davis and Burns [18] introduced the UUnifast-Discard algorithm. This approach accommodates multiprocessor constraints by repeatedly invokes UUniFast and rejects any generated vector containing an individual utilization greater than one. However, this discard-based strategy compromises tractability in specific configurations. For example, in an 8-processor system ($m = 8$) with a target total utilization of $U = 8$, as the number of tasks n decreases from 16 to 9, the probability of generating a valid vector plummets. The algorithm must discard an extremely large number of invalid vectors before yielding a single valid sequence, rendering the process computationally intractable. Attempting to impose tighter constraints to correct this behavior only exacerbates the rejection rate, worsening the tractability bottleneck.

2.3.3 RandFixedSum

The RandFixedSum algorithm, originally developed by Stafford [19] and later popularized in the real-time community by Emberson et al. [20], efficiently generates utilization vectors for any feasible combination of n and U under symmetric per-task lower and upper bounds. This algorithm relies on symmetric constraints, typically setting individual task utilization boundaries to $u^{\max} = 1$ and $u^{\min} = 0$. While highly efficient under these conditions, RandFixedSum's mathematical formulation depends heavily on this geometric symmetry. As a result, directly

extending the method to non-symmetric constraints would be computationally intractable for specific tasks.

2.3.4 Dirichlet-Rescale (DRS)

To overcome the limitations of traditional methods, the Dirichlet-Rescale (DRS) algorithm proposed by Griffin et al. [16] provides a general-purpose solution with the following signature:

$$u = \mathrm{DRS}(n, U, u^{\max}, u^{\min}) \tag{2.10}$$

where $\sum_i^n u_i^{\min} \leq U \leq \sum_i^n u_i^{\max}$; $0 \leq u_i^{\min} \leq u_i^{\max}$ and $u = (U_1, U_2, ..., U_n)$ represents the generated utilization vector n is the cardinality of the task set, U is the target total utilization. The parameters $u^{\max} = (U_1^{\max}, U_2^{\max}, ..., U_n^{\max})$ and $u^{\min} = (U_1^{\min}, U_2^{\min}, ..., U_n^{\min})$ define the maximum and minimum utilization constraints for each task, respectively, defaulting to $u^{\max} = u^1$ and $u^{\min} = u^0$ when omitted.

The DRS algorithm is designed to satisfy all three criteria: efficiency, uniformity, and flexibility. It enables fine-grained control over individual task utilization boundaries, allowing researchers to easily restrict a subset of tasks (e.g., enforcing $U_i \leq 0.5$ for $i \leq 5$). Crucially, DRS can be applied recursively in scenarios involving multi-valued task utilizations or complex sub-components.

This makes it an exceptionally practical tool for constructing realistic datasets in advanced scheduling domains, such as mixed-criticality systems and multi-core platforms. However, recent literature [21] has shown that DRS does not guarantee a uniform distribution in all constrained settings, particularly when multiple constraints may be violated simultaneously. Nevertheless, at most 10,000 task sets are generated per configuration in this study, whereas the reported deviation was only detectable over a sample of 10^7 vectors; a sample of our size is therefore unlikely to be affected. Moreover, since all algorithms under comparison are evaluated on identical task sets, any bias introduced by the generator affects them equally and does not compromise the validity of the relative comparison. DRS thus serves as the primary algorithm selected for dataset generation in this study.

TAIPEI TECH

Chapter 3 Methodology

This chapter presents the proposed methodology for elastic scheduling on uniprocessor and multiprocessor systems. Section 3.1 introduces the overall workflow and the roles of the Improved Elastic Scheduling, Hybrid Breakpoint Search, and LB-Enhanced Binary Search algorithms. Section 3.2 presents the corresponding pseudocode and explains each algorithm line by line.

3.1 Main Method

Figure 3.1 illustrates the architectural framework of the proposed Low-Overhead Breakpoint Partitioned Earliest Deadline First (LOBP-EDF) methodology. In contrast to designs that switch based on task set scale, this framework selects the compression algorithm based on the system's current operating mode, leveraging the complementary characteristics of the two algorithms in terms of decision latency and compression quality:

- Emergency Mode: When the system detects an overload or when a high-criticality task experiences an execution time overrun requiring an immediate mode switch, the Hybrid Breakpoint Search algorithm is deployed. In this mode, decision latency is the primary consideration, and a moderate degree of over-compression is acceptable. As discussed in [14], [15], in mixed-criticality systems, extending the periods of low-criticality tasks (even if the compression amount is slightly higher than necessary) remains superior to the traditional approach of directly discarding their jobs.
- Normal Mode: During routine admission control, offline resource allocation, or non-emergency dynamic adjustments, the LB-Enhanced Binary Search algorithm is utilized. In this mode, compression quality is the primary focus, and the resulting λ value is highly consistent with that of the original method.

The schedulability decisions of both modes are completely identical, hence, mode switching does not compromise the system's feasibility guarantees. The operational divergence lies solely

in the trade-off between the compression amount and decision latency. The operating mode is designated by an upper-layer scheduler or a mode manager, as this framework is not responsible for mode detection.

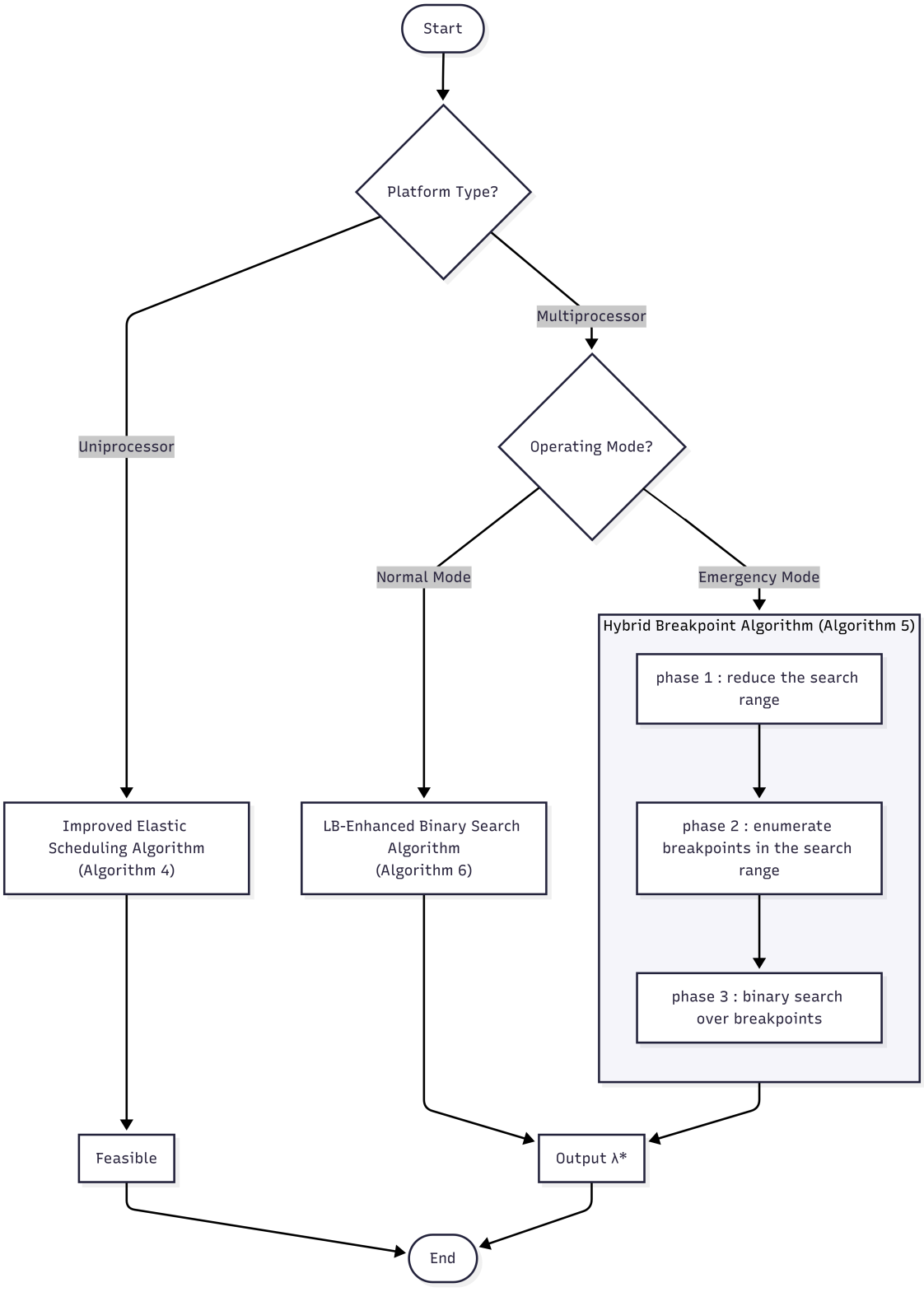


Figure 3.1: Flowchart of the proposed LOBP-EDF framework

Additionally, the specific operational phases within the Hybrid Breakpoint Search algorithm are discussed in comprehensive detail in Section 3.1.2.

3.1.1 Improved Elastic Scheduling Algorithm

Algorithm 2 repeatedly performs division operations during the task-clamping process. Since floating-point division is generally more computationally expensive than multiplication, the proposed method transforms the original decision condition into an equivalent form. Applying this concept of substitution, we improve the decision condition in Algorithm 2:

$$U_i^{\max} - \frac{U_{\text{Sum}} - (U_d - \Delta)}{E_{\text{Sum}}} \times E_i \leq U_i^{\min} \tag{3.1}$$

Rearranging the terms yields:

$$U_i^{\max} - U_i^{\min} \leq \frac{U_{\text{Sum}} - (U_d - \Delta)}{E_{\text{Sum}}} \times E_i \tag{3.2}$$

Assuming that $E_{\text{Sum}} > 0$, multiplying both sides by E_{Sum} results in a new inequality:

$$(U_i^{\max} - U_i^{\min}) \times E_{\text{Sum}} \leq (U_{\text{Sum}} - (U_d - \Delta)) \times E_i \tag{3.3}$$

Assuming that $E_i > 0$, Equation (3.3) can be rewritten as:

$$\frac{U_i^{\max} - U_i^{\min}}{E_i} \times E_{\text{Sum}} \leq (U_{\text{Sum}} - (U_d - \Delta)) \tag{3.4}$$

We arrive at the final simplified condition, using the breakpoint definition $\Phi_i = (U_i^{\max} - U_i^{\min})/E_i$, the clamping condition becomes:

$$\Phi_i E_{\text{Sum}} \leq (U_{\text{Sum}} - (U_d - \Delta)) \tag{3.5}$$

By applying strength reduction, the proposed method replaces repeated divide-multiply evaluations in the task-clamping condition with multiplication and comparison operations involving the precomputed breakpoint Φ_i. The algebraic equivalence of this transformation follows from the assumptions that $E_i > 0$ and $E_{\text{Sum}} > 0$. The task list is traversed in ascending order of Φ_i, so that tasks reaching their minimum utilization are identified one after another, which removes unnecessary comparisons.

TAIPEI TECH

Additional code-level optimizations are applied to reduce execution overhead. Common-subexpression elimination(CSE) is used to cache repeatedly accessed loop bounds and data references, thereby reducing redundant address calculations. Loop-Invariant Code Motion(LICM) is also applied to move invariant ratio and availability calculations outside the inner loop. Consequently, the number of division operations in each outer-loop iteration is reduced from $n - i$ to one. These transformations preserve the feasibility of the decision of Algorithm 2 while reducing its arithmetic overhead. The resulting procedure is presented in Algorithm 4.

3.1.2 Hybrid Breakpoint Search Algorithm

For Algorithm 3, the most direct improvement is to enumerate candidate linear-region crossover breakpoints $O(n^2)$ breakpoints, sort them, and remove duplicates to perform a binary search over this discrete set. By avoiding the use of the ε variable, this approach significantly reduces the number of comparisons, allowing the algorithm to rapidly identify a feasible candidate value of λ within the selected allocation-heuristic framework. However, the enumeration and sorting cost $O(n^2 \log n)$ under large task sets scales rapidly with n. As demonstrated by the experiments in Chapter 4, as n increases, the enumeration overhead grows rapidly, diminishing the efficiency gains achieved by reducing the search iterations. This observation suggests that the efficiency of the breakpoint method arises from searching within a discrete set of structural change points, whereas its primary overhead stems from enumerating all change points across the entire continuous range. These two aspects can be treated separately. If the search space can be narrowed to a smaller interval at a low cost prior to enumeration, it becomes necessary to enumerate only the breakpoints falling within the reduced interval need to be retained and examined. This suppresses the enumeration overhead while fully preserving the advantages of discrete search. Therefore, based on this observation, this study proposes the “Hybrid Breakpoint Search algorithm,” which combines the rapid interval reduction of numerical binary search with breakpoint-based discrete search.

TAIPEI TECH

As illustrated in Figure 3.2, the Hybrid Breakpoint Search algorithm divides the search process into three distinct phases:

- Phase 1 (Range Reduction): The binary-search procedure of Algorithm 3 is executed for K iterations over the initial interval $[0, \lambda^{\max}]$. In each iteration, the algorithm calculates the midpoint of the current interval and performs a schedulability check. If the task set is schedulable, the upper bound is shrunk to the midpoint; otherwise, the lower bound is raised to the midpoint. After K iterations, the search space is compressed to the interval $[\lambda_{\text{rang_lo}}, \lambda_{\text{rang_hi}}]$, the resulting interval has a width $\frac{\lambda^{\max}}{2^K}$ of the original range.
- Phase 2 (Breakpoint Enumeration): Breakpoints within the compressed interval are enumerated based on two principal types of candidate structural change points:
  - Clamping breakpoint φ_i: for each task τ_i with $E_i > 0$, the clamping breakpoint is defined as $\varphi_i = (U_i^{\max} - U_i^{\min})/E_i$ when λ reaches φ_i, the utilization τ_i reaches $U_i^{\min}$. For all $\lambda \geq \varphi_i$, the task remains clamped at its minimum utilization.
  - Ordering breakpoint $\lambda_{\text{cross}}(i, j)$: An ordering breakpoint is any value of λ within the reduced search interval that satisfies $U_i(\lambda) = U_j(\lambda)$. At such a point, the non-increasing utilization used by BFD or FFD may change.
- Phase 3 (Discrete Search): The discrete candidate set generated in Phase 2 is sorted and deduplicated. If monotonicity is established for the selected schedulability predicate, binary search is applied to the sorted candidate set. Otherwise, the candidates must be evaluated sequentially to avoid skipping a feasible transition.

The complete procedure is presented in Algorithm 5.

TAIPEI TECH

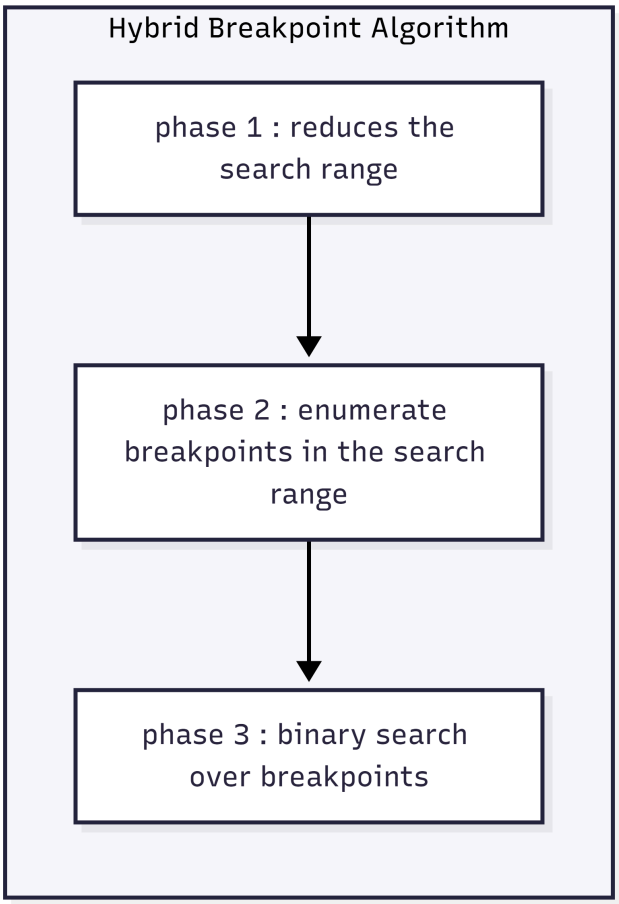


Figure 3.2: Flow Chart of Hybrid Breakpoint Search Algorithm

Here, we discuss the selection of the parameter K in Phase 1. Since the optimal value of K is closely tied to the scale of the task set, its configuration must be balanced. When n is small, the overhead of the $O(n^2)$ enumeration is negligible; an excessively large K would only introduce redundant schedulability checks in Phase 1, thereby degrading efficiency. Conversely, when n is large, the enumeration overhead becomes dominant, requiring a larger K to effectively minimize the number of candidate breakpoints in Phase 2. Through extensive experimental tuning, the parameter K controls the trade-off between Phase 1 schedulability-check overhead and the number of candidate breakpoints retained for Phase 2. The experiment configuration used in this study sets $K = 4$ for $n < 32$ and $K = 6$ for $n \geq 32$. These values were selected empirically based on the experiments reported in Chapter 4 and are not claimed to be universally

optimal. Note that the selection of K is an internal parameter of the Hybrid Breakpoint Search algorithm and is independent of the framework-level operating-mode decision.

Now let's discuss the computational complexity of the Hybrid Breakpoint Search algorithm. Let p denote the number of candidate breakpoints retained within the reduced search interval $[\lambda_{\text{range_lo}}, \lambda_{\text{range_hi}}]$. A partitioned-EDF schedulability check requires $O(n \log n + \text{nm})$ time, including task ordering and allocation to m processor cores. Therefore, Phase 1, which performs K schedulability checks, requires $O(K(n \log n + \text{nm}))$ time. In phase 2, generating the clamping breakpoints requires $O(n)$ time, while examining all task pairs for ordering breakpoints requires $O(n^2)$ time. Sorting and removing duplicates from the p retained candidate breakpoints requires $O(p \log p)$ time. Phase 3 performs a binary search over the sorted candidate set, requiring $O(\log p)$ schedulability checks and therefore $O(\log p(n \log n + \text{nm}))$ time. Consequently, the overall time complexity of the Hybrid Breakpoint Search algorithm is $O((K + \log p)(n \log n + \text{nm}) + n^2 + (p \log p))$. Since $p = O(n^2)$ in the worst case, the complexity can be expressed as $O((K + \log n)(n \log n + \text{nm}) + n^2 \log n)$. Although Phase 1 does not reduce the worst-case $O(n^2)$ cost of examining all task pairs, it typically reduces the number p of candidate breakpoints retained for sorting and discrete search, thereby lowering the practical execution overhead.

3.1.3 LB-Enhanced Binary Search Algorithm

The original Algorithm 3 conducts its search within the $[0, \lambda^{\max}]$ interval with a target precision of ε, requiring $\log(\lambda^{\max}/\varepsilon)$ rounds. To shrink this search space, one potential approach is to incorporate the concept mentioned at the end of Section 2.2.2, which compresses the U_d of Algorithm 2 to the $\frac{m+1}{2}$ safety boundary. However, this bound is overly conservative: it yields minimal contraction on the right half of the interval while adding computational overhead, so retaining $\lambda^{\max}$ as the upper bound is the more practical choice. The left half, by contrast, offers greater opportunity. When the system is overloaded, the minimum λ^* typically lies far from 0, and a substantial number of search iterations are wasted merely confirming the unschedulability

of this region. Pruning it at low cost can therefore significantly reduce the number of search rounds.

Partitioned EDF admits a task set only when its aggregate utilization stays within the available core count; formally, $\sum U_i(\lambda) \leq m$. Since $\sum U_i(\lambda)$ is a piecewise linear non-increasing function of λ, λ_{LB} is set to the smallest λ at which the summed utilization first drops to or below m. It can be analytically computed using the procedure underlying Algorithm 4 by setting the target utilization to $U_d = m$. Any deviation from the exact value arises solely from floating-point arithmetic, as the derivation itself involves no approximation. Because $\sum U_i(\lambda) \leq m$ is necessary but not sufficient, any $\lambda < \lambda_{\mathrm{LB}}$ inevitably causes the total utilization to exceed m, rendering the task set unschedulable.

The minimum feasible compression parameter must therefore satisfy $\lambda^* \geq \lambda_{\mathrm{LB}}$. Thus, using λ_{LB} as the initial lower bound excludes only values that are necessarily infeasible and does not remove the minimum feasible solution from the search interval. The LB-Enhanced Binary Search algorithm retains the complete structure of Algorithm 3, replacing only the lower search bound with λ_{LB}. Under the same target precision ε, the search range narrows to $[\lambda_{\mathrm{LB}}, \lambda^{\max}]$, reducing the required number of rounds from $\lceil \log_2(\lambda^{\max}/\varepsilon) \rceil$ to $\lceil \log_2\left(\frac{\lambda^{\max}-\lambda_{\mathrm{LB}}}{\varepsilon}\right) \rceil$. Then the resulting overall time complexity is $O\Big(n \log n + \lceil \log_2\left(\frac{\lambda^{\max}-\lambda_{\mathrm{LB}}}{\varepsilon}\right) \rceil (n \log n + \mathrm{nm})\Big)$.

3.2 Pseudocode and Detailed Algorithm Description

We first introduce the Improved Elastic Scheduling Algorithm 4, which is designed as an enhancement over Algorithm 2.

Algorithm 4: Improved_Elastic_Scheduling

function IMPROVED_ELASTIC_SCHEDULING(Γ, U_d)
    Input:A list Γ of elastic tasks sorted in non-decreasing order of their Φ_i parameters and a desired utilization U_d
    Output: A feasibility status and the updated task set Γ
    $U_{\mathrm{SUM}} \leftarrow 0$
    $E_{\mathrm{SUM}} \leftarrow 0$
    for $\tau_i \in \Gamma$ **do**

```
    U_SUM = U_SUM + U_i^max
    E_SUM = E_SUM + E_i
  end
  n ← |Γ|                                              ▷ CSE
  avail ← U_d − Δ                                      ▷ LICM
  for i = 1 to n do
    τ ← Γ[i]                                           ▷ CSE
    if Φ_i × E_SUM ≤ (U_SUM − avail) then
      τ.u_i ← τ.u_i^min
      avail ← avail − τ.u_i^min
      if avail < 0 then
        return Infeasible
      end
      U_SUM ← U_SUM − τ.u^max
      E_SUM ← E_SUM − τ.e_i
    else
      ratio ← (U_SUM − avail) / E_SUM                  ▷ LICM
      for j = i to n do
        τ_j ← Γ[j]                                     ▷ CSE
        τ_j.u_j ← τ_j.U^max − ratio × τ_j.E_j
      end
      return Feasible,Γ
    end
  end
  return Feasible,Γ
end
```

Algorithm 4 receives an over-utilized elastic task set Γ—already ordered by ascending Φ_i—together with a target total utilization U_d. It performs utilization compression to evaluate the schedulability of the task set and computes the newly compressed utilization U_i for each task. In the implementation, the lines labeled with CSE (Common Subexpression Elimination) or LICM (Loop Invariant Code Motion) represent source-level code optimizations. The former (such as caching the size using `size_t n = Gamma.size();` or defining reference types) allows the CPU to calculate the memory address of `Gamma[i]` only once and cache it, thereby eliminating substantial redundant address calculation overhead. The latter (LICM) manually hoists

TAIPEI TECH

loop-invariant variables out of the loop body. Specifically, the avail variable avoids redundant calculations of ratio in the else branch, while the ratio variable itself hoists the division operation out of the inner loop so that it is executed only once.

Lines 5-10 correspond to the initialization phase, which declares the accumulated variables U_{SUM} and E_SUM, summing up each task's U_i^{MAX} (representing the ideal maximum utilization demand of the task) and its elasticity coefficient E_i. Lines 13-30 execute the utilization compression phase. When the conditional check in Line 15 (representing equation 3.5) evaluates to true, tasks transitioning into the Γ_{Fixed} state are locked, and their values are subtracted from U_{SUM} and E_SUM to update the system parameters. Lines 16-19 verify whether the minimum required utilization of the task set exceeds the specified utilization target U_d, which is normally set to 1 for uniprocessor feasibility analysis and to m when deriving the multiprocessor lower bound. If this limit is exceeded, it indicates that the task set is unschedulable even when all tasks are compressed to their minimum utilization bounds. Finally, Lines 24-29 start with task i perform the actual utilization compression and dynamically update the utilizations for all tasks remaining in the Γ_{VARIABLE} state. Once the inner loop completes and the compression process finishes successfully, the algorithm returns feasible to indicate schedulability.

Next, we describe the Hybrid Breakpoint Search Algorithm (Algorithm 5) in detail:

Algorithm 5: Hybrid Breakpoint Search

```
1:  function HYBRID BREAKPOINT SEARCH(Γ, m)
2:      λ^max ← lambdaMax (Γ)
3:      n ← |Γ|
4:      ▷ Phase 1: applying binary search reduces the search range
5:      if Γ(0) is schedulable on m cores then return FEASIBLE, 0
6:      if Γ(λ^max) is not schedulable on m cores then return HEURISTIC_FAILURE, ⊥
7:      rang_lo ← 0
8:      rang_hi ← λ^max
9:      if n < 32 then
10:         K ← 4
11:     else
12:         K ← 6
```

```
    end
    for i = 1 to K while rang_hi − rang_lo > ε do
        mid ← (rang_lo + rang_hi)/2
        if Schedulable(Γ, m, mid) then
            rang_hi ← mid
        else
            rang_lo ← mid
        end
    end
    ▷ Phase 2: enumerate breakpoints in [rang_lo, rang_hi]
    cands ← {rang_lo}
    for each τ_i ∈ Γ do
        if rang_lo < φ_i < rang_hi then
            cands ∪ {φ_i}
        end
    end
    for each pair τ_i, τ_j ∈ Γ with i < j do
        if e_i ≠ e_j then
            lc ← (τ_{u_i^max} − τ_{u_j^max}) / (e_i − e_j)
            if rang_lo < lc < rang_hi then
                cands ∪ {lc}
            end
        end
    end
    cands ∪ {rang_hi}
    ▷ Phase 3: binary search over breakpoints
    Sort cands and remove duplicates
    lo ← 0
    hi ← |cands| − 1
    while lo < hi do
        mid ← lo + ⌊(hi − lo)/2⌋
        if Schedulable(Γ, m, cands[mid]) then
            hi ← mid
        else
            lo ← mid + 1
        end
    end
    return FEASIBLE, cands[hi]
```

TAIPEI TECH

51: **end**

Algorithm 5 identifies the minimum feasible candidate value of λ within the enumerated breakpoint set under the selected allocation heuristic. Initially, Line 2, the $\lambda^{\max}$ function is invoked to determine the maximum possible lambda value for the input task set. Line 3 declares and assigns the cardinality of the task set $|\Gamma|$ to n. Lines 5-6 perform preliminary boundary checks on the task set: specifically, verifying whether it is schedulable at $\mathrm{lambda} = 0$, or remains unschedulable even at the maximum bound lambdaMax. Lines 7-8 declare and initialize the search interval boundaries, range_lo and range_hi, for the initial binary search phase. Lines 9-13 configure the range-reduction iteration parameter K: setting it to 4 for smaller task sets where $n < 32$, and to 6 for larger task sets. Lines 14-21 then execute the range-reduction phase using the binary search framework from Algorithm 3 to compress the search interval. Line 23 declares the candidate breakpoint array cands and initializes it by adding the compressed lower boundary range_lo. Lines 24-28 identify boundary breakpoints τ_{φ_i} within the compressed range—representing the lambda values where each task i reaches its minimum utilization lower bound U_min—and append them to cands. Lines 29-36 identify and append crossover sorting breakpoints "lc" that fall within the compressed boundaries to cands. Line 37 then appends the upper boundary range_hi to the end of the cands array. Line 39 sorts the candidate compression values in non-decreasing order and removes duplicates before binary search. Lines 40-41 declare and initialize the index pointers, lo and hi, for the discrete search phase. Finally, Lines 42-49 perform a binary search over the discrete candidate set cands to identify the minimum feasible lambda value of candidate set, note that the discrete binary search assumes that the schedulability predicate of the selected allocation heuristic is monotonic over the ordered candidate values. Upon exiting the loop, Line 50 returns the final result.

Finally, we present the LB-Enhanced Binary Search Algorithm (Algorithm 6):

Algorithm 6: LB-Enhanced Binary Search

1: **function** LB-ENHANCED BINARY SEARCH(Γ, m)

TAIPEI TECH

2: **Input:**A list Γ of elastic tasks to schedule on m processor cores
3: **Output:** An approximation of the minimum feasible compression parameter λ within
4: tolerance ε, under the selected partitioning heuristic.
5: $\lambda^{\max} \leftarrow 0$
6: **for** $\tau_i \in \Gamma$ **do**
7: $\lambda_i^{\max} \leftarrow \frac{U_i^{\max} - U_i^{\min}}{E_i}$
8: $\lambda^{\max} \leftarrow \max(\lambda^{\max}, \lambda_i^{\max})$
9: **end**
10: **if** $\Gamma(0)$ is schedulable on m cores **then return** 0
11: **if** $\Gamma(\lambda^{\max})$ is **not** schedulable on m cores **then return** HEURISTIC_FAILURE, $\perp$
12: $\lambda_{\mathrm{HI}} \leftarrow \lambda^{\max}$
13: $\lambda_{\mathrm{LB}} \leftarrow$ Compute_Elastic_Lower_Bound $(\Gamma$,m$)$
14: **if** $\Gamma(\lambda_{\mathrm{LB}})$ is schedulable on m cores **then return** λ_{LB}
15: $\lambda_{\mathrm{LO}} \leftarrow \lambda_{\mathrm{LB}}$
16: **while** true **do**
17: $\lambda \leftarrow (\lambda_{\mathrm{HI}} + \lambda_{\mathrm{LO}})/2$
18: **if** $\Gamma(\lambda)$ is schedulable on m cores **then** $\lambda_{\mathrm{HI}} \leftarrow \lambda$
19: **else** $\lambda_{\mathrm{LO}} \leftarrow \lambda$
20: **if** not $(\lambda_{\mathrm{HI}} - \lambda_{\mathrm{LO}} > \varepsilon)$ **then**
21: break
22: **end**
23: **end**
24: **return** λ_{HI}
25: **end**

Algorithm 6 computes, for an elastic task set Γ, the least lambda that keeps the system feasible on m cores. Since its fundamental structural framework is identical to Algorithm 3, we focus our description strictly on the modified component. Specifically, in Line 13, the declaration and initialization of the λ_{LO} variable are modified to invoke the analytical lower bound procedure from Algorithm 4. Line 14 checks the λ_{LB} indicates whether the values can be scheduled on m cores; if the condition is satisfied, λ_{LB} is returned. Utilizing the returned value of this invocation as the optimized lower bound completes the proposed enhancement for Algorithm 6.

TAIPEI TECH

Chapter 4 Experiment and Results

To set up the experimental environment, Section 4.1 lists the requirements, Section 4.2 details each step of the environment setup, and Section 4.3 presents the dataset generation and experimental evaluation.

4.1 Requirements

Table 4.1 lists the software requirements. Raspberry Pi OS is used as the operating system for the Raspberry Pi. As the officially designated OS for the Raspberry Pi, it is built upon the stable and reliable Debian Linux, inheriting all of Linux's core advantages while providing a complete graphical interface and a rich ecosystem of associated software tools. Moreover, its high degree of openness and excellent kernel extensibility allow the experiments in this thesis to bypass user-space restrictions by writing and compiling custom kernel modules and hardware drivers. The algorithms are implemented in C++, and execution-time performance is quantified by measuring the elapsed processor cycles, while dataset generation and the plotting of experimental results are implemented in Python. The code is compiled under the GCC C++17 standard. For Algorithm 4, the optimization level, the -O0 is used to prevent unexpected instruction reordering from interfering with the reading of the cycle counter, which would otherwise compromise the accuracy of the results; for Algorithm 5 and 6, whose measurements are less susceptible to such interference, the -O2 is used.

Table 4.1: Software Requirements

Software Type	Software Name	Version
Operating system	Raspberry Pi OS	64-bit [22]
Programming Language	C++	Debian 14.2.0-19 [23]
Programming Language	Python	3.12.6 [24]

Table 4.2 lists the third-party software requirements. C++ third-party libraries are used for container construction, statistical data handling, and timing within the code, and the DRS [25] tool released by David Griffin et al. [16] is used to generate the experimental datasets.

Table 4.2: Third-Party Software Requirements

Name	Version	License
Dirichlet-Rescale (DRS) Algorithm [25]	v2.0.1	MIT License [26]

Table 4.3 Detailed Hardware Specifications of the Raspberry Pi 5 Development Board [27].

Table 4.3: Hardware Requirements

Hardware Type	Name	Detail
CPU	Broadcom BCM2712 quad-core ARM Cortex A76 processor	Running at 2.4 GHz
RAM	LPDDR4X-4267 SDRAM	8 GB
CACHES	-	512KB per-core L2 caches and a 2MB shared L3 cache

4.2 Environment Setup

This subsection describes the environment setup. First, the dedicated Raspberry Pi Imager software is downloaded to flash the operating system onto an SD card, following the setup steps illustrated on the left side of Figure 4.1. Once the flashing process is complete, the SD card is inserted into the card slot of the Raspberry Pi 5 board.

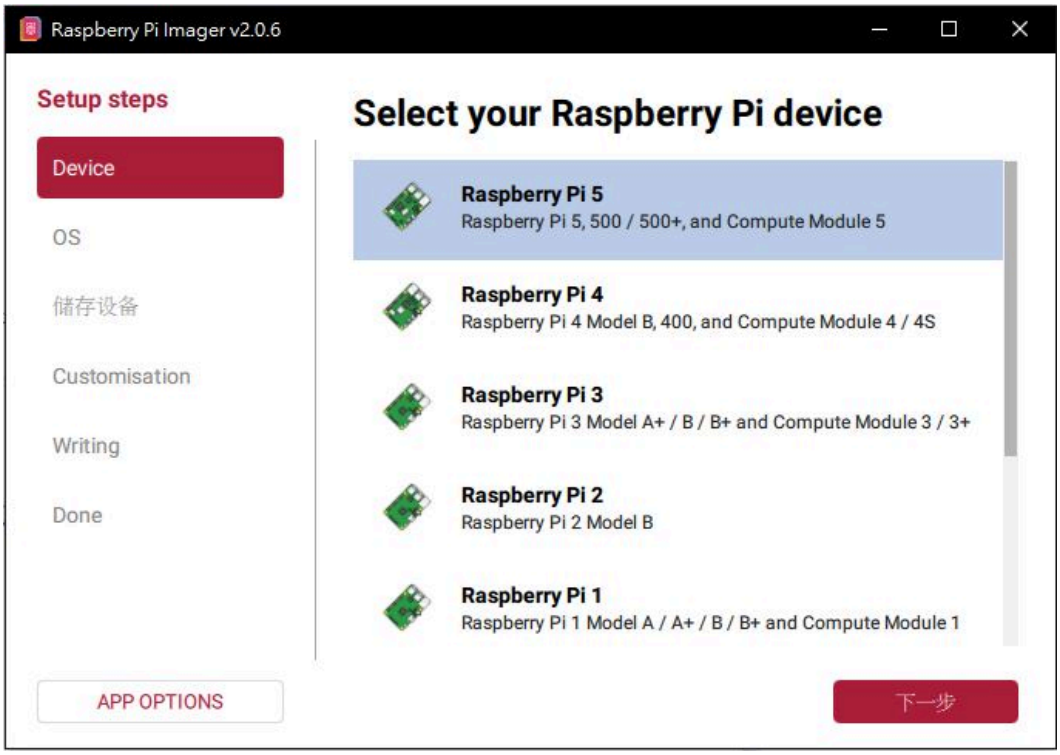


Figure 4.1: Raspberry Pi Imager Interface

Next, at the system level, we boot the Raspberry Pi and perform the OS installation and user configuration. We then configure the Raspberry Pi through the command line, disabling specific features and isolating a designated core to prevent interference during program measurement. As described in [28], we append `isolcpus=3` to the end of `/boot/firmware/cmdline.txt` to isolate the fourth core. Following [29], we then disable Real-Time Throttling, a feature that allows a real-time task to fully utilize the CPU without being arbitrarily suspended.

```
echo -1 > /proc/sys/kernel/sched_rt_runtime_us
```

The following command runs the target program on the isolated CPU 3 and sets its `SCHED_FIFO` priority to 99:

```
sudo taskset -c 3 chrt -f 99 ./algorithm
```

The cpufrequtils [30] toolkit is used to set the Linux kernel's CPUFreq governor to performance, which locks the CPU at its maximum clock frequency of 2.4 GHz.

```
sudo apt-get install -y cpufrequtils
sudo cpufreq-set -g performance
```

Next, referring to the ARM Architecture Reference Manual - Performance Monitors [31] and The Linux Kernel Module Programming Guide [32], we write the driver and kernel module: `pmu_cycles.hpp`, `pmu_enable.c`, and a `Makefile` (used to compile pmu_enable.c into a kernel module). This module allows the ARM Performance Monitoring Unit (PMU) to be accessed from user space in order to read the cycle counter for measurement in the subsequent experiments.

`pmu_cycles.hpp`

```
static inline uint64_t read_cycles(){
  uint64_t val;
  asm volatile("isb; mrs %0, pmccntr_el0" : "=r"(val) :: "memory");
  return val;
}
static inline uint64_t elapsed_cycles(uint64_t start, uint64_t end){
```

```
    return end - start;
  }
```

pmu_enable.c

```
  static void enable_pmu_on_cpu(void *info){
    u64 val;
    asm volatile("msr pmuserenr_el0, %0" :: "r"(0xFULL));
    val = 0ULL;
    val |= (1ULL << 0);
    val |= (1ULL << 1);
    val |= (1ULL << 2);
    val |= (1ULL << 6);
    asm volatile("msr pmcr_el0, %0" :: "r"(val));
    asm volatile("msr pmcntenset_el0, %0" :: "r"(1ULL << 31));
    asm volatile("isb");
  static int __init pmu_init(void){
    on_each_cpu(enable_pmu_on_cpu, NULL, 1);
    return 0;
  }
  static void __exit pmu_exit(void){}
  module_init(pmu_init);
  module_exit(pmu_exit);
```

makefile

```
  obj-m += pmu_enable.o
  KDIR ?= /lib/modules/$(shell uname -r)/build
  PWD  := $(shell pwd)
  all:
    $(MAKE) -C $(KDIR) M=$(PWD) modules
  clean:
    $(MAKE) -C $(KDIR) M=$(PWD) clean
```

We install the kernel headers, then compile and load the kernel module to unlock the hardware restrictions.

TAIPEI TECH

```
sudo apt install raspberrypi-kernel-headers
sudo insmod pmu_enable.ko
```

Finally, we simply include the header file in the program to be measured, at which point the entire environment setup is complete.

4.3 Experiment

To enable a comparison with the original algorithm, this subsection generates the experimental datasets and designs the experiments following the methodology of Sudvarg et al. [4], [5].

4.3.1 Task Sets Generation

The procedure for generating the task sets for the real-time scheduling experiments is as follows:

For the uniprocessor algorithm, we generate sets Γ of tasks τ:

- We consider n=2-50 tasks and generate 10000 sets for each. Every set contains a total maximum utilization $U_{\text{SUM}}^{\max}$ in the range $(1.0, 2.0]$ and a total minimum utilization $U_{\text{SUM}}^{\min}$ in the range $(0.0, 1.0]$, both generated randomly using `np.random.uniform()`.
- For each set, DRS [16] is used to distribute the parameters $U_{\text{SUM}}^{\max}$ and $U_{\text{SUM}}^{\min}$ across the $U_i^{\max}$ and $U_i^{\min}$ of every τ_i in the set in an unbiased random manner, ensuring that each $U_i^{\min}$ does not exceed its corresponding $U_i^{\max}$.
- Each τ_i is assigned an elastic coefficient E_i, generated randomly within the range $(0, 1]$ using `np.random.uniform()`.

For the multiprocessor algorithm, we generate sets Γ of tasks τ:

- consider different multiprocessor platforms m =4, 8, 16 processor cores, for each number of processor cores m consider $n = (2m, 4m, 8m)$ tasks.
- For each task configuration, we generate 1000 sets (formed from the combination of m, n, α, and u), where $\alpha \in \{0.6, 0.8, 1.0\}$ denotes the upper limit when assigning the maximum

utilization $U_i^{\max}$ to each τ_i, and $u \in \{1.1, 1.5, 1.9\}$ determines the total maximum utilization $U_{\text{SUM}}^{\max}$ of each set (formed from $u \times m \times \alpha$).

- Likewise, DRS is used to distribute the total maximum utilization $U_{\text{SUM}}^{\max}$ of each set across the $U_i^{\max}$ of every τ_i, while the minimum utilization $U_i^{\min}$ in the range $(0, U_i^{\max}]$ and E_i in the range $(1, 5]$ are generated randomly using `np.random.uniform()`.

4.3.2 Experimental Evaluation

We first conduct experiments on Algorithm 4 and its comparison algorithms, measuring the elapsed processor cycles of their init and compression phases separately—using the driver developed in Section 4.2—to quantify execution-time performance. When measuring Algorithm 1, its init phase involves computing $U_{\text{SUM}}^{\min}$ and checking whether it exceeds U_d, while its compression phase corresponds to the while loop in the algorithm. For Algorithm 2, the original authors [4], [5] provide three methods of constructing the data structure; here we select the best-performing one, the array, for comparison. Its init phase includes computing U_{SUM} and E_{SUM}, computing the φ_i of each task, and the time to sort them, while its compression phase is the for loop that iterates according to the parameter φ_i. Finally, for our improved Algorithm 4, the measured components are the same as those of Algorithm 2. For the three algorithms above, we measure the mean, median, and maximum according to Section 4.3.1, with the experimental data shown in Figure 4.2.

TAIPEI TECH

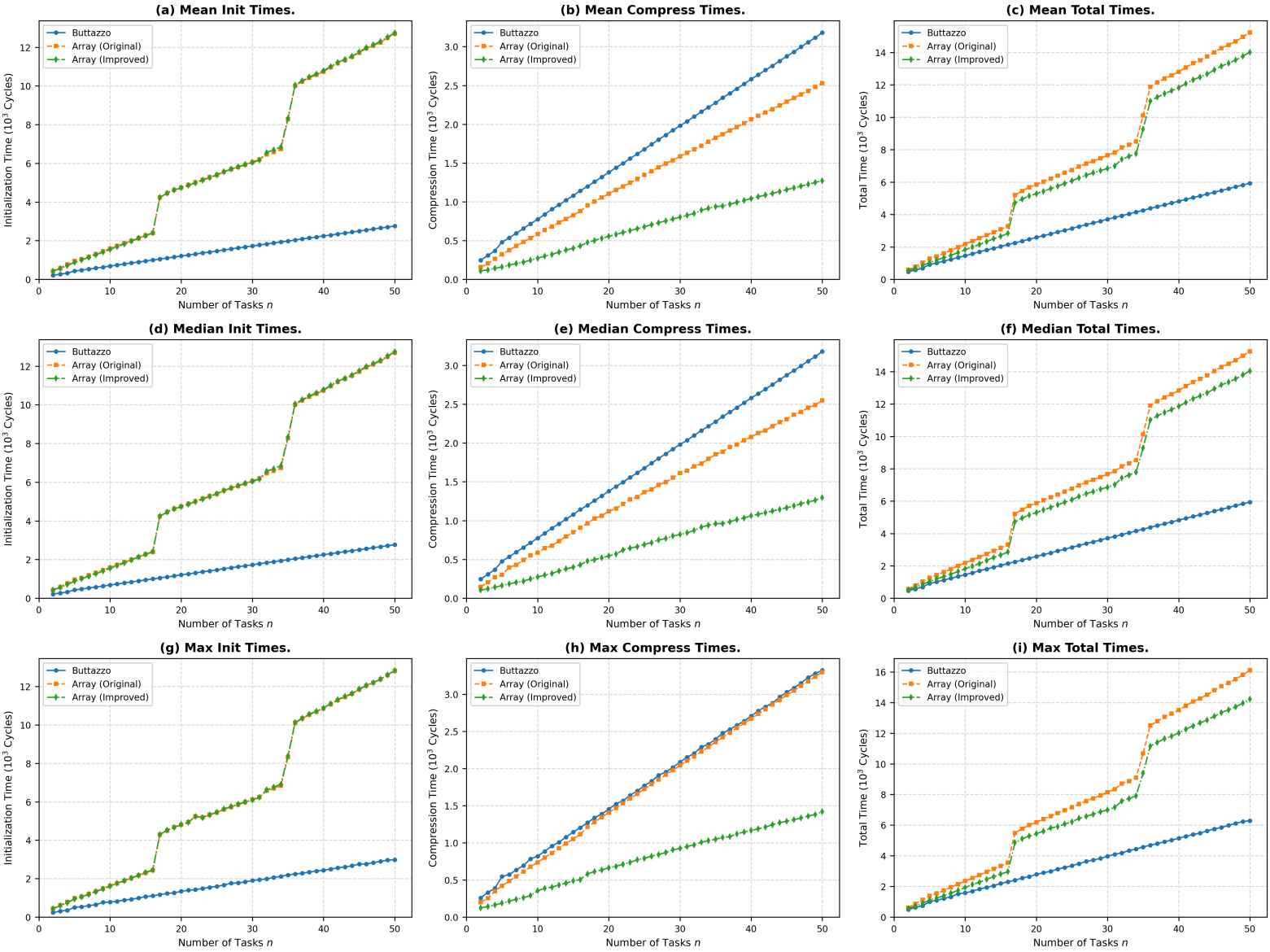


Figure 4.2: Uniprocessor Algorithms Result Comparison

The results can be discussed in the following respects:

- As observed from Figure 4.3, the improved algorithm achieves a 43.8% compression time speedup at the initial N=2, which gradually increases until it stabilizes at a twofold speedup after N=5. Table 4.4 presents its result data under the largest task size (N=50). As described in Section 3.1.1, this method successfully improves the execution time by up to twofold.
- Based on the experiments in [5] (specifically Sections 3.2.4 and 3.2.5), since we did not modify the original method but only optimized its computational part, our experimental results are equally applicable to the comparisons in the original work. For Task Admission, the improved version holds an even greater advantage over Algorithm 1; for Online Adaptation, the original work uses a real-world automotive benchmark, whose results show that the speedup achieved by the algorithm is of practical relevance—and by the same reasoning, our improved version likewise holds an advantage.

TAIPEI TECH

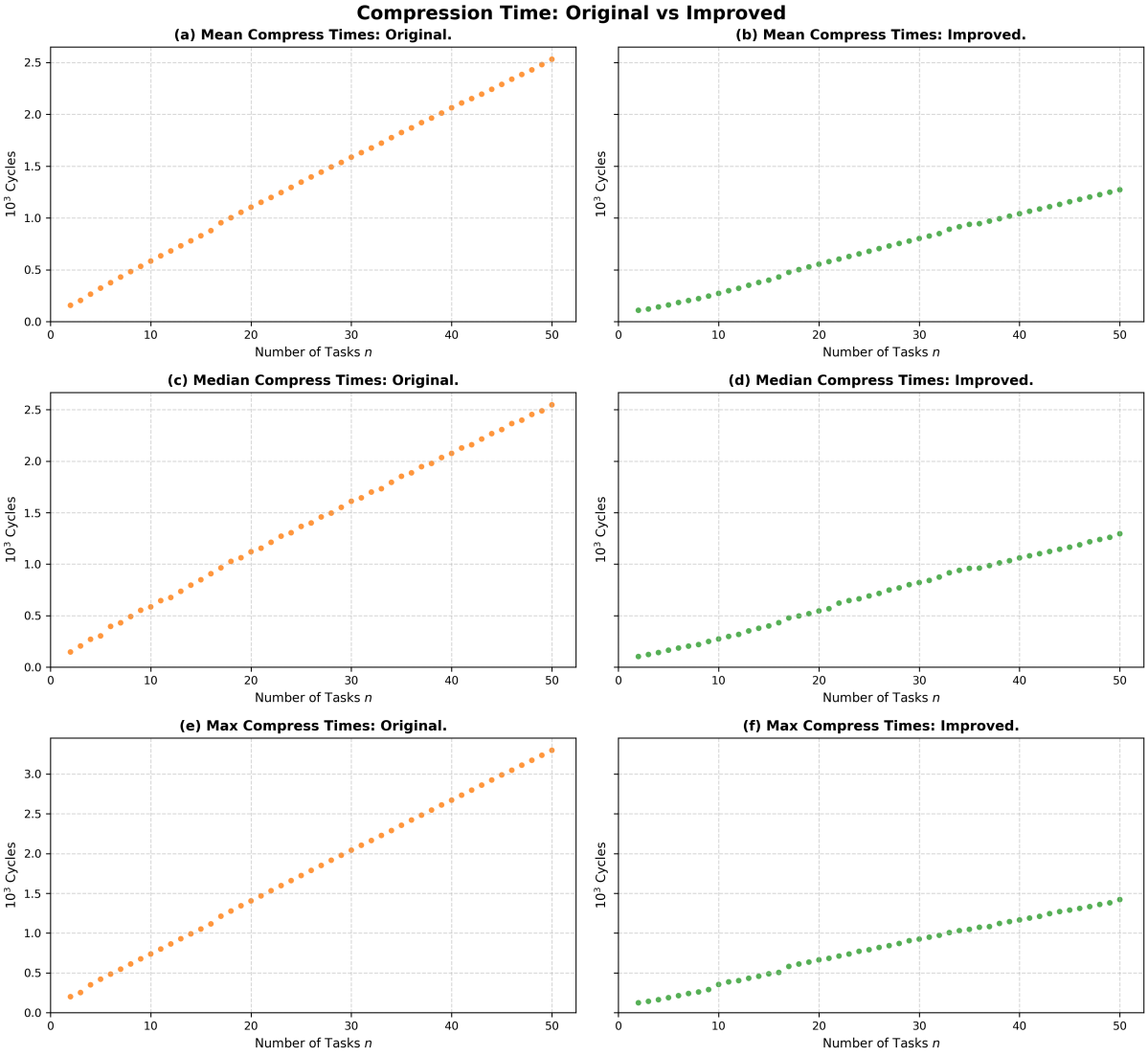


Figure 4.3: Uniprocessor Algorithms Compression Time Comparison

Table 4.4: Mean, median, and maximum execution cycles of uniprocessor algorithms during the compression phase (N=50)

	Buttazzo	**Sudvarg**	**Algorithm 4 (Ours)**
MEAN	3180	2531	1271
MEDIAN	3177	2547	1295
MAXIMUM	3321	3298	1421

Next, we conduct experiments on Algorithms 5 and 6, comparing four methods: the pure breakpoint algorithm, the two algorithms under the LOBP-EDF framework, and the original binary search (Algorithm 3).

For Algorithm 3, we set its ε to $\frac{\lambda^{\max}}{1000}$ (the same setting as in the original work [4], [5]), and then test each λ value using the best-fit decreasing and first-fit decreasing bin-packing heuristics until the algorithm returns its final result. For our pure breakpoint algorithm, Algorithm 5, and Algorithm 6 (whose ε also follows the original work's setting), we test using the same bin-packing heuristics. To ensure a fair comparison, all four algorithms use exactly the same schedulability check, and the execution-time measurement interval covers only the algorithm itself, excluding file output and statistical computation.

The experimental results are shown in Figure 4.4, Figure 4.5, and Figure 4.6, which provides a basis for analyzing each approach below:

- For the pure breakpoint algorithm, as shown by the red dashed line in the figures, it is faster than the original binary search across all test combinations, and its advantage widens as n increases (at $m = 16, n = 128, \alpha = 0.6, u = 1.9$, BP requires about 8.72×10^5 cycles, whereas Original requires about 23.62×10^5 cycles, a speedup of about 2.71x). This result validates the effectiveness of the discrete-search concept: since the size of the breakpoint candidate set is $O(n^2)$, performing a binary search over it requires only $O(\log n^2)$ schedulability checks, fewer than the $\log\left(\frac{\lambda^{\max}}{\varepsilon}\right)$ checks required by a continuous search. However, its enumeration and sorting cost of $O(n^2 \log n)$ accumulates rapidly as n grows (at $n = 128$, the number of candidate points can reach 8128), diminishing part of the speedup benefit—an observation that motivated the subsequent design of the Hybrid version.
- For BP_Hybrid, it first narrows the search range using K rounds of binary search, and then enumerates the breakpoints within the narrow interval. As shown by the green dashed line in the figures, this strategy further reduces the enumeration cost: at $m = 16, n = 128, \alpha = 0.6, u = 1.9$, BP_Hybrid requires about 4.13×10^5 cycles, a further reduction of about 53% from the 8.72×10^5 of pure BP, and a speedup of about 5.72x over Original. This improve-

ment appears consistently across all test combinations, making BP_Hybrid the fastest of the four methods.

- For the LB-Enhanced Binary Search, as shown by the orange dashed line, it roughly coincides with the blue Original: the two are nearly identical under low-load combinations, but a stable improvement is observed under high-load combinations (at $m = 16, n = 128, \alpha = 1.0, u = 1.9$, it is 20.62×10^5 versus Original's 21.60×10^5, an improvement of about 12.7%). This phenomenon reflects that the tightness of the analytical lower bound λ_{LB} is correlated with the system load: the higher the load, the farther λ^* is from 0, and the larger the search space that the lower bound can exclude. In addition, computing λ_{LB} itself incurs an $O(n \log n)$ sorting cost, which partially offsets the saved checks. In summary, the BP-series methods hold a significant advantage in execution speed, while the speedup of LB-Enhanced, though more moderate, is stable. However, execution speed is only one aspect of the evaluation; the differences among the methods in compression quality, and the causes behind these differences, are discussed next.

TAIPEI TECH

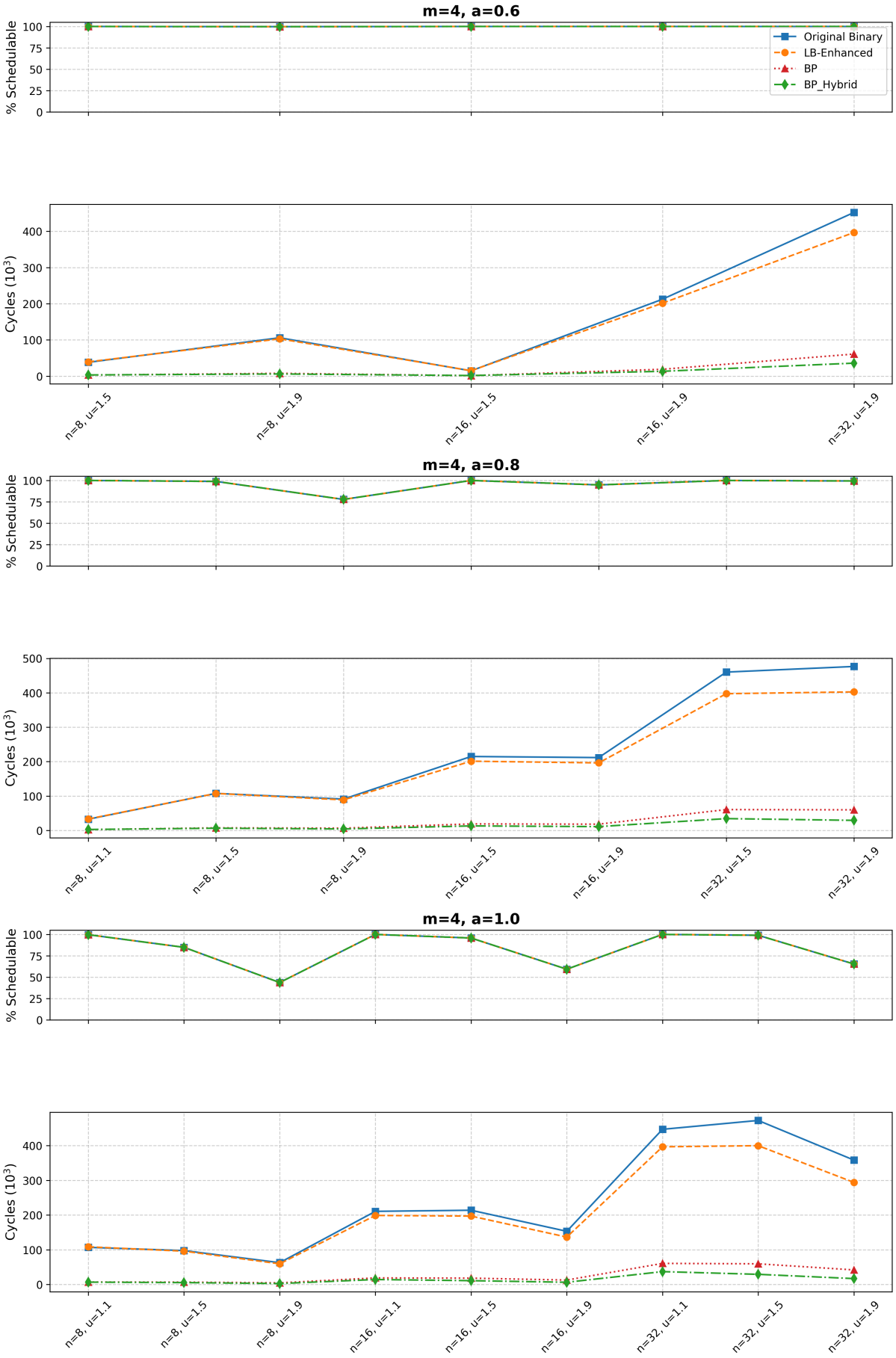


Figure 4.4: Multiprocessor Algorithms Comparison M=4

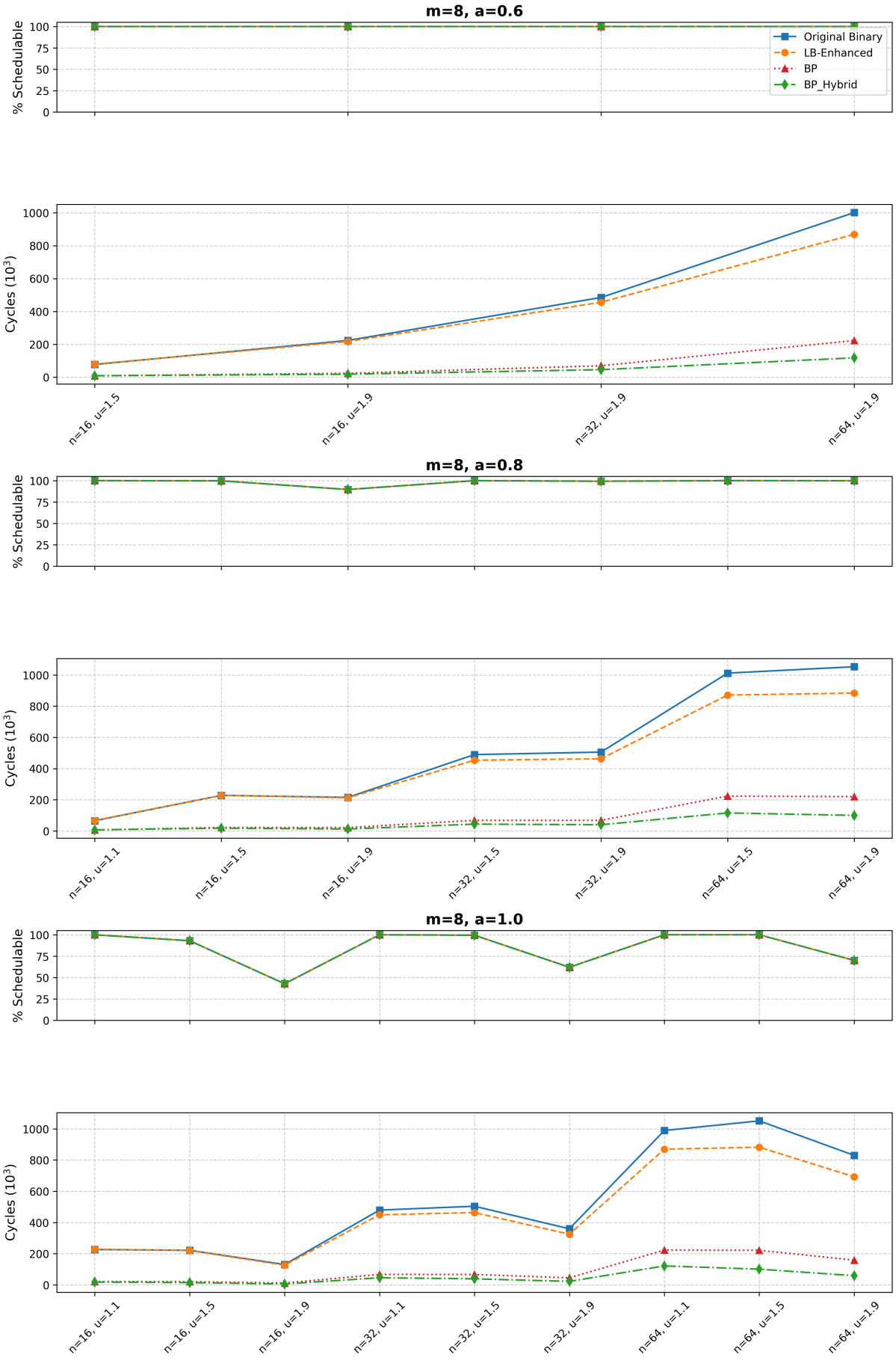


Figure 4.5: Multiprocessor Algorithms Comparison M=8

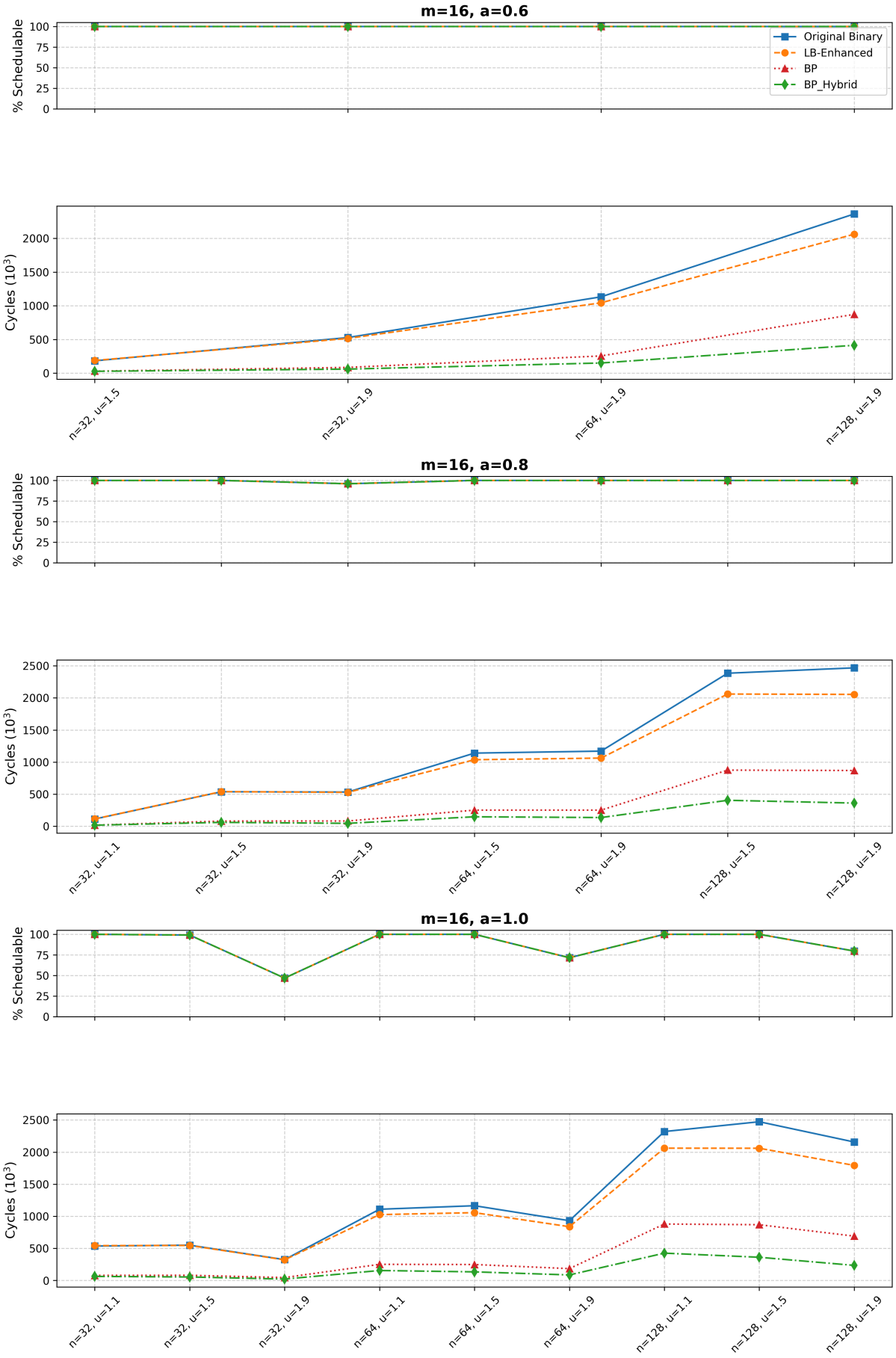


Figure 4.6: Multiprocessor Algorithms Comparison M=16

Beyond execution performance, the λ values returned by the methods also differ, with the results shown in Figure 4.7 and Figure 4.8.

- Since LB-Enhanced preserves the complete structure of the original binary search and only raises the search lower bound from 0 to λ_{LB}, its λ values are highly consistent with those of Original: across all three core-count configurations, 99.9% of the task sets have their $|\mathrm{diff}|$ falling within ε ($m = 4$: 16773/16782, $m = 8$: 17101/17118, $m = 16$: 17388/17410), with an average $|\mathrm{diff}|/\varepsilon$ of only about 0.30. This result confirms that λ_{LB}, used as the search lower bound, does not exclude the correct solution, and that its speedup does not come at the cost of compression quality.
- Since BP_Hybrid searches over a discrete candidate set, its output differs more markedly from Original, and the proportion of this difference is correlated with the core count: at $m = 4$, 76.9% of the task sets have their $|\mathrm{diff}|$ exceeding ε, dropping to 50.7% at $m = 8$ and 22.2% at $m = 16$. This trend stems from the definition of ε: since $\varepsilon = \lambda^{\max}/1000$ and $\lambda^{\max}$ increases with the core count, the same absolute error is normalized and thus magnified at smaller m. It is also worth noting that the differences are almost entirely positive, indicating that BP_Hybrid tends toward over-compression (returning a larger λ than Original); moreover, for a fixed m, the distributions narrow and approach 0 as n increases. Notably, this accuracy trend runs opposite to the performance trend described earlier—BP_Hybrid achieves its most significant speedup at $m = 4$, yet also incurs its greatest accuracy loss at this configuration. This trade-off forms the empirical basis for the LOBP-EDF framework's selection of algorithms according to the operating mode.

Despite the differences in λ values, the schedulability decisions of the three methods are entirely consistent: the feasibility decisions of both LB-Enhanced and BP_Hybrid match those of Original for every task set, without exception. This result means that the differences among the methods lie only in "how much to compress," not in "whether it can be scheduled," so the mode switching of LOBP-EDF does not affect the system's feasibility guarantee.

Further analyzing the direction of the differences reveals a key phenomenon: in most task

sets, the λ returned by LB-Enhanced is smaller than that of Original ($m = 4$: 54.8%, $m = 8$: 52.9%, $m = 16$: 49.0%). This phenomenon warrants deeper examination—LB-Enhanced and Original use exactly the same binary search structure, the same ε, and the same bin-packing heuristics, the only difference being that the search lower bound is raised from 0 to λ_{LB}. Under the monotonicity assumption of Theorem 2, the two should converge to the same boundary; the fact that more than half of the task sets violate this inference shows that monotonicity does not hold under heuristic decisions. This is precisely the phenomenon predicted by Corollary 6 of [5]: when schedulability is determined by a bin-packing heuristic rather than an exact test, the schedulability of $\Gamma(\lambda)$ does not guarantee that of $\Gamma(\lambda')$ for all $\lambda' > \lambda$. In the non-monotonic region, different search paths converge to different local feasibility boundaries.

TAIPEI TECH

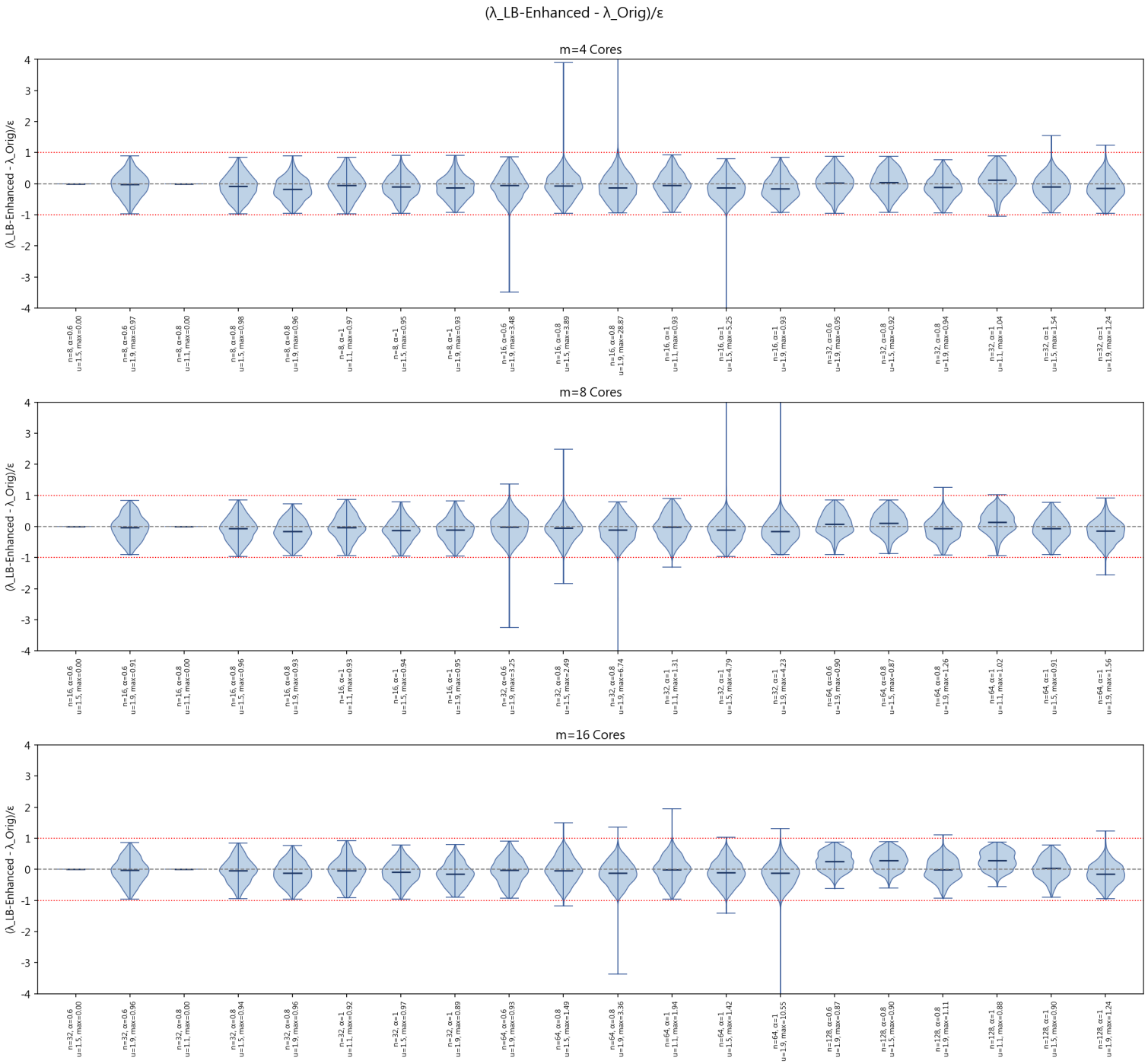


Figure 4.7: Lambda difference between LB-Enhanced and Original Binary

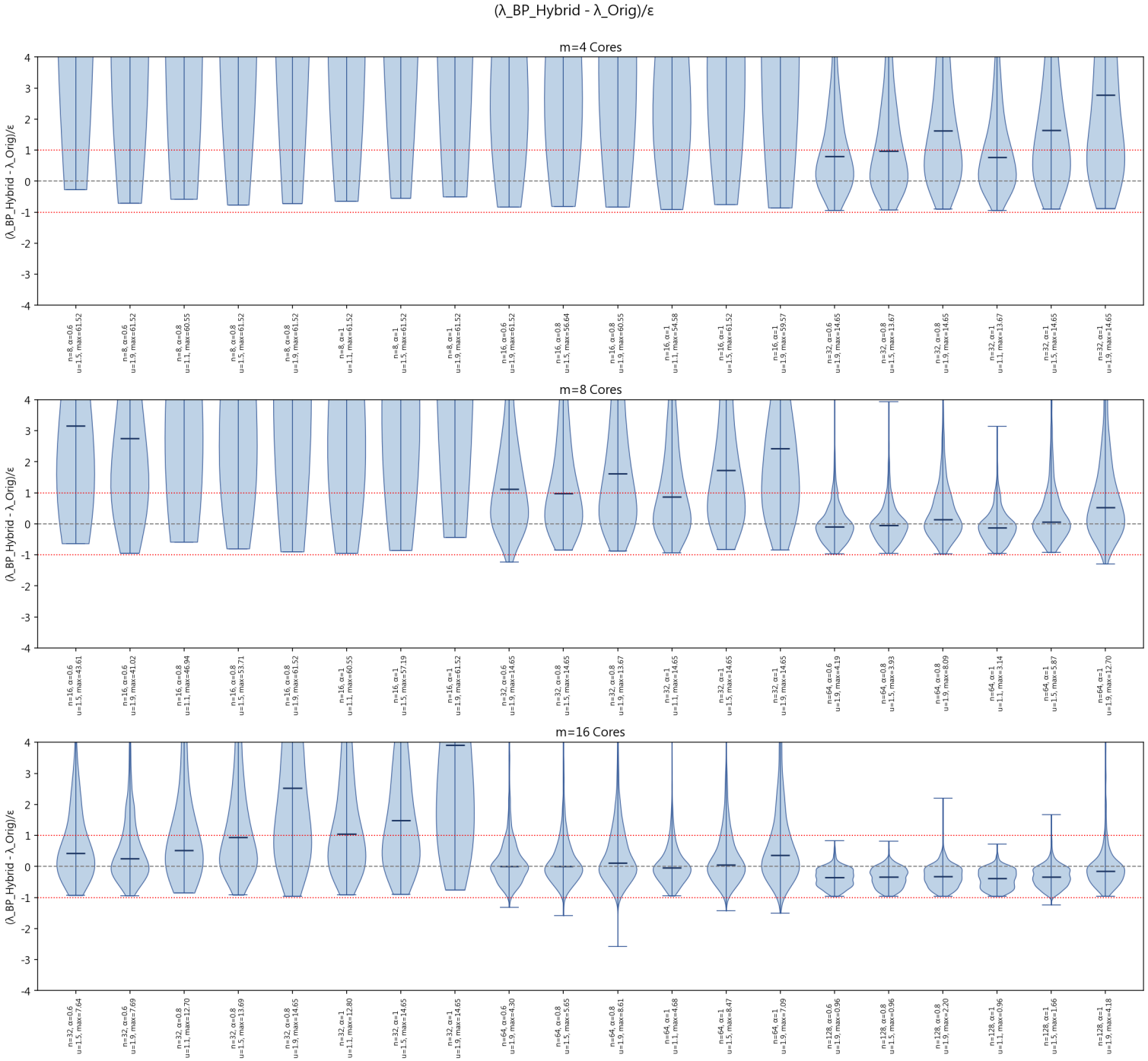


Figure 4.8: Lambda difference between BP_Hybrid and Original Binary

Chapter 5 Conclusion and Future Work

5.1 Conclusion

This study builds upon the elastic scheduling model proposed by Buttazzo et al. [1], addressing its two limitations—the quadratic time complexity of the original algorithm and its restriction to uniprocessors—and extends the work of Sudvarg et al. [4], [5] on the uniprocessor quasilinear solution and the multiprocessor binary search, proposing two improvements:

- On the uniprocessor side, this study optimizes the quasilinear-time algorithm and proposes Algorithm 4. The experimental results (Figure 4.3, Table 4.4) show that the execution speed of its compression phase improves by 43.8% at N=2 and stabilizes at an approximately twofold speedup after N=5; under the largest task size of N=50, its mean cycles drop from the baseline's 2531 to 1271. Since the improvement acts only on the computational part without altering the logic of the original method, this speedup applies equally to the Task Admission and Online Adaptation comparison scenarios in the original work, giving it practical application value.
- On the multiprocessor side, this study proposes the Low-Overhead Breakpoint Partitioned Earliest Deadline First (LOBP-EDF) framework, centered on a binary-search-based breakpoint method and offering two modes according to the operating scenario. Emergency Mode adopts the Hybrid Breakpoint Search algorithm, which first narrows the range using several rounds of binary search and then enumerates the breakpoints; under the high-load combination of m=16, n=128, it achieves a speedup of up to about 5.72x over Original, making it the fastest of the four methods, at the cost of more noticeable over-compression when the core count is small (m=4). Normal Mode adopts the LB-Enhanced Binary Search, which leverages the analytical properties of the uniprocessor algorithm to derive a lower bound λ_{LB} for the compression parameter, thereby narrowing the search range; although its speedup is more moderate (about 12.7% under high load), it is stable, and across all three core-count configu-

TAIPEI TECH

rations 99.9% of the task sets have their λ difference falling within ε, making its compression quality nearly identical to that of Original.

Most importantly, the schedulability decisions of both modes match those of Original for every task set, without exception. This means that the differences between the modes lie only in how much to compress, not in whether it can be scheduled; therefore, LOBP-EDF enables the system to trade off between decision speed and compression quality according to its real-time requirements, without sacrificing the feasibility guarantee. Moreover, this study also observes that in more than half of the task sets the λ returned by LB-Enhanced is smaller than that of Original, confirming the phenomenon predicted by Corollary 6 of [5]—that when schedulability is determined by a bin-packing heuristic rather than an exact test, the monotonicity assumed by the original method (Theorem 2) does not hold.

5.2 Future Work

Although this study achieves substantial advancements in execution performance, several directions remain worth extending:

- Reducing the breakpoint enumeration cost: The candidate set of the pure breakpoint method has a size of $O(n^2)$, and its enumeration and sorting cost of $O(n^2 \log n)$ accumulates rapidly as n grows (reaching 8128 candidate points at n=128), diminishing the net performance gains. Future work could explore aggressive candidate-point generation and proactive pruning strategies, or use incremental data structures to avoid re-sorting each time, so as to further reduce the overhead under large-scale task sets.
- Strengthening the tightness of the lower bound: The speedup of LB-Enhanced is correlated with the system load—the higher utilization yields a tighter analytical lower bound λ_{LB} which in turn prunes a larger segment of the infeasible search space, but its benefit is limited under low load, and the $O(n \log n)$ sorting cost of computing λ_{LB} itself partially offsets the savings. Future work could study load-aware lower-bound derivation, or combine it with an upper-

TAIPEI TECH

bound estimate for two-sided convergence, this would ensure a robust and stable performance dividend across all operating workloads.

- Automated mode switching: Currently, the selection between Emergency and Normal mode relies on manual specification. A compelling extension would involve designing a dynamic, runtime strategy capable of automatically switching modes based on the system's real-time state, such as the current load and the available decision-time budget. Mixed-criticality systems, in which elastic compression has been used to determine task suspension across a criticality mode switch [15], present a natural and fertile testing ground for evaluating such automated policies.
- Extension to broader scheduling scenarios: This study focuses on partitioned EDF with the best-fit and first-fit decreasing bin-packing heuristics. Although polynomial-time exact elastic algorithms for global EDF and global RM have recently been established [5], the breakpoint and lower-bound concepts proposed here have not yet been applied under those paradigms, nor to semi-partitioned scheduling or to other bin-packing heuristics. Extending the model beyond implicit deadlines is likewise an open direction: recent work has addressed sporadic constrained-deadline elastic tasks [7], [8], particularly in evaluating whether a comparable discrete breakpoint structure exists within those complex domains.
- Real-system deployment validation: The current evaluation relies heavily on simulation frameworks and processor-cycle measurement. A critical next step is to integrate LOBP-EDF into a commercial or open-source real-time operating system (RTOS) running on a heterogeneous multi-core platform to validate its end-to-end benefits under dynamic overload adaptation with real-world workloads.

TAIPEI TECH

References

[1] G. C. Buttazzo, G. Lipari, and L. Abeni, “Elastic task model for adaptive rate control,” in *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, Madrid, Spain: IEEE, 1998, pp. 286–295. doi: 10.1109/REAL.1998.739754.

[2] J. Orr and S. Baruah, “Multiprocessor scheduling of elastic tasks,” in *Proceedings of the 27th International Conference on Real-Time Networks and Systems (RTNS)*, Toulouse, France: ACM, 2019, pp. 133–142. doi: 10.1145/3356401.3356403.

[3] J. Orr and S. Baruah, “Algorithms for implementing elastic tasks on multiprocessor platforms: a comparative evaluation,” *Real-Time Systems*, vol. 57, no. 1, pp. 227–264, 2021, doi: 10.1007/s11241-020-09358-9.

[4] M. Sudvarg, C. Gill, and S. Baruah, “Improved implicit-deadline elastic scheduling,” in *Proceedings of the IEEE International Symposium on Industrial Embedded Systems (SIES)*, Chengdu, China: IEEE, 2024, pp. 50–57. doi: 10.1109/SIES62473.2024.10768003.

[5] M. Sudvarg, C. Gill, and S. Baruah, “Improved Elastic Scheduling Algorithms for Implicit-Deadline Tasks,” *Leibniz Transactions on Embedded Systems (LITES)*, vol. 10, no. 2, pp. 2:1–2:36, 2025, doi: 10.4230/LITES.10.2.2.

[6] M. Sudvarg, C. Gill, and S. Baruah, “Linear-Time Admission Control for Elastic Scheduling,” *Real-Time Systems*, vol. 57, no. 4, pp. 485–490, 2021, doi: 10.1007/s11241-021-09373-4.

[7] S. Baruah, “Improved Uniprocessor Scheduling of Systems of Sporadic Constrained-Deadline Elastic Tasks,” in *Proceedings of the 31st International Conference on Real-Time Networks and Systems (RTNS)*, Dortmund, Germany: ACM, 2023, pp. 67–75. doi: 10.1145/3575757.3575759.

[8] W. Liu, J.-J. Chen, and Y. Yang, “Schedulability Analysis and Performance Optimization for Constrained-Deadline Elastic Tasks,” in *Proceedings of the 32nd International Con-*

TAIPEI TECH

ference on Real-Time Networks and Systems (RTNS), Porto, Portugal: ACM, 2025, pp. 207–217. doi: 10.1145/3696355.3696362.

[9] J. M. López, J. L. Díaz, and D. F. García, "Utilization bounds for EDF scheduling on real-time multiprocessor systems," *Real-Time Systems*, vol. 28, no. 1, pp. 39–68, 2004, doi: 10.1023/B:TIME.0000033378.56741.14.

[10] S. Baruah, "Partitioned EDF scheduling: a closer look," *Real-Time Systems*, vol. 49, no. 6, pp. 715–729, 2013, doi: 10.1007/s11241-013-9186-0.

[11] M. Sudvarg, J. Buhler, R. D. Chamberlain, C. Gill, J. Buckley, and W. Chen, "Parameterized Workload Adaptation for Fork-Join Tasks with Dynamic Workloads and Deadlines," in *Proceedings of the 2023 IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Niigata, Japan: IEEE, 2023, pp. 232–242. doi: 10.1109/RTCSA58653.2023.00035.

[12] M. Sudvarg, D. Wang, J. Buhler, and C. Gill, "Subtask-Level Elastic Scheduling," in *Proceedings of the 2024 IEEE Real-Time Systems Symposium (RTSS)*, York, UK: IEEE, 2024, pp. 388–401. doi: 10.1109/RTSS62706.2024.00040.

[13] J. Orr, J. C. Uribe, C. Gill, S. Baruah, K. Agrawal, S. J. Dyke, A. Prakash, I. J. Bate, C. Wong, and S. Adhikari, "Elastic Scheduling of Parallel Real-Time Tasks with Discrete Utilizations," in *Proceedings of the 28th International Conference on Real-Time Networks and Systems (RTNS)*, Paris, France: ACM, 2020, pp. 117–127. doi: 10.1145/3394810.3394824.

[14] H. Su and D. Zhu, "An Elastic Mixed-Criticality task model and its scheduling algorithm," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Grenoble, France: IEEE, 2013, pp. 147–152. doi: 10.7873/DATE.2013.043.

[15] Z. Sun, M. Sudvarg, and C. Gill, "Elastic Scheduling for Graceful Degradation of Mixed-Criticality Systems," in *Proceedings of the 32nd International Conference on Real-Time Networks and Systems (RTNS)*, Porto, Portugal: ACM, 2025, pp. 218–228. doi: 10.1145/3696355.3699701.

TAIPEI TECH

[16] D. Griffin, I. Bate, and R. I. Davis, "Generating utilization vectors for the systematic evaluation of schedulability tests," in *Proceedings of the 2020 IEEE Real-Time Systems Symposium (RTSS)*, Houston, TX, USA: IEEE, 2020, pp. 76–88. doi: 10.1109/RTSS49844.2020.00018.

[17] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1–2, pp. 129–154, 2005, doi: 10.1007/s11241-005-0507-9.

[18] R. I. Davis and A. Burns, "Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems," *Real-Time Systems*, vol. 47, no. 1, pp. 1–40, 2011, doi: 10.1007/s11241-010-9106-5.

[19] R. Stafford, "Random vectors with fixed sum." Accessed: Jul. 22, 2026. [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/9700-random-vectors-with-fixed-sum

[20] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *Proceedings of the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, Brussels, Belgium, 2010, pp. 6–11. Accessed: Jul. 22, 2026. [Online]. Available: https://www.cs.york.ac.uk/rts/static/papers/R:Emberson:2010a.pdf

[21] R. S. H. Willemsen, W. van den Heuvel, and M. van de Velden, "Generating Random Vectors satisfying Linear and Nonlinear Constraints." 2025. doi: 10.48550/arXiv.2501.16936.

[22] "Raspberry Pi OS." Accessed: Jul. 22, 2026. [Online]. Available: https://www.raspberrypi.com/software/operating-systems/

[23] "Debian Package: libgcc-14-dev (14.2.0-19)." Accessed: Jul. 22, 2026. [Online]. Available: https://packages.debian.org/trixie/libgcc-14-dev

[24] "Python 3.12.6." Accessed: Jul. 22, 2026. [Online]. Available: https://www.python.org/downloads/release/python-3126/

[25] "DRS algorithm github." Accessed: Jul. 22, 2026. [Online]. Available: https://github.com/dgdguk/drs

TAIPEI TECH

[26] “DRS algorithm License.” Accessed: Jul. 22, 2026. [Online]. Available: https://github.com/dgdguk/drs/blob/master/LICENSE.txt

[27] “Raspberry Pi 5 product brief.” Accessed: Jul. 22, 2026. [Online]. Available: https://pip-assets.raspberrypi.com/categories/892-raspberry-pi-5/documents/RP-008348-DS-6-raspberry-pi-5-product-brief.pdf

[28] “The kernel’s command-line parameters.” Accessed: Jul. 22, 2026. [Online]. Available: https://www.kernel.org/doc/html/v4.14/admin-guide/kernel-parameters.html

[29] “Real-Time group scheduling.” Accessed: Jul. 22, 2026. [Online]. Available: https://docs.kernel.org/scheduler/sched-rt-group.html

[30] “Debian Packages: cpufrequtils.” Accessed: Jul. 22, 2026. [Online]. Available: https://packages.debian.org/sid/cpufrequtils

[31] “D13.1 About the Performance Monitors.” Accessed: Jul. 22, 2026. [Online]. Available: https://developer.arm.com/documentation/ddi0487/mb/-Part-D-The-AArch64-System-Level-Architecture/-Chapter-D13-The-Performance-Monitors-Extension/-D13-1-About-the-Performance-Monitors?lang=en

[32] “The Linux Kernel Module Programming Guide..” Accessed: Jul. 22, 2026. [Online]. Available: https://sysprog21.github.io/lkmpg/

TAIPEI TECH